

Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor

Department of Computer Science

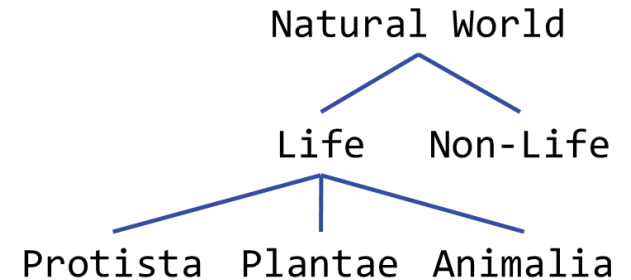
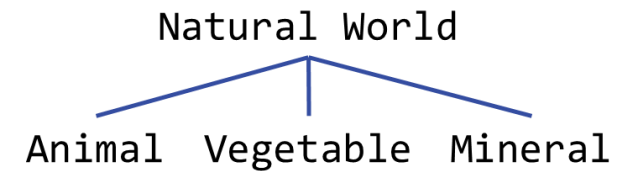
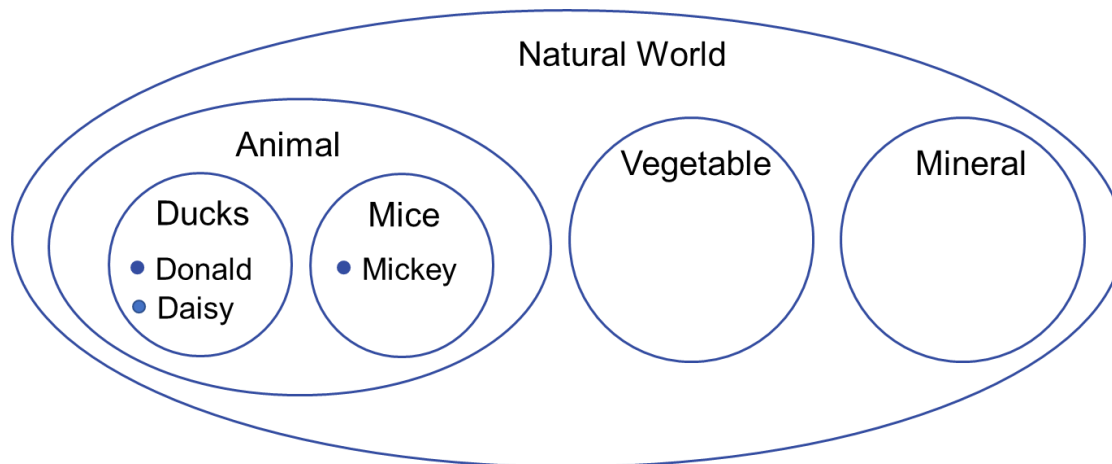
Cornell University

Classes and Objects

A *taxonomy* is a system of classification. Taxonomies are an essential mechanism for organizing subject matter.

Hierarchical taxonomies in which concepts are organized into tree structures are ubiquitous. In a hierarchy, the most general concept is placed at the root of the tree, and subordinate concepts branch out from there.

Each category is a set of individuals. A Venn diagram depicts categories as nested regions, and individuals as dots.



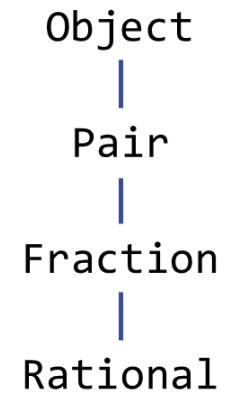
Taxonomic categories in programming are called *classes*, and the individuals of a class are its *objects*. The root category is **object**.

We illustrate classes and objects by implementing `Pair`, `Fraction`, and `Rational`. Every rational is a fraction, and every fraction is a pair of integers, and every pair is an **object**.

We then implement `ArrayList[E]`, a parameterized class for representing and manipulating collections of type `E` elements. We use the class `ArrayList[Rational]` to complete code for enumerating rationals.

Because we can define `HashSet[E]`, class similar to `ArrayList[E]`, it is easy to replace one with the other, and compare their speed. We do so, and demonstrate the dramatic speedup of hash tables over lists.

Finally, in a bit of a double cross, we observe that collections weren't ever actually needed for enumerating rationals in the first place, and obtain a still-faster implementation without them.



A *class* is a collection of variable declarations and method definitions. An *object* is a dynamic instantiation of the variables (and methods) of a class whose declarations (and definitions) are not declared and initialized at the top-level of the class.

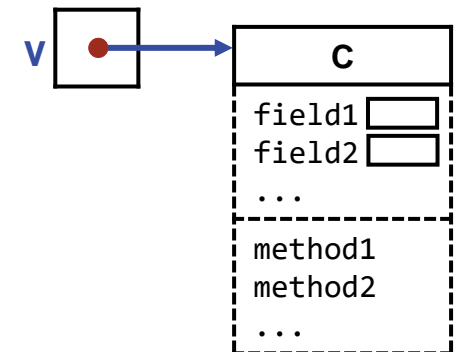
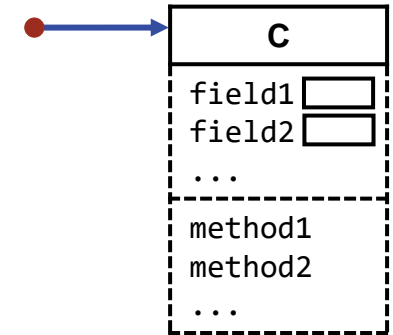
Such variables are known as *object fields* or *instance variables* (and such methods are known as *instance methods*). Objects and references to them are depicted as shown.

Classes are *types*. If *C* is a class, a variable *v* of *type C* is obtained by executing the declaration:

v: *C* = *expression*

That is, variable *v* (with type *C*) is initialized with the value of the *expression*.

Such a variable can hold a reference to an object of type *C*.



An object o of type C is created by executing the expression

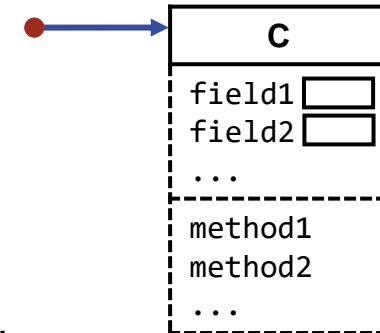
$C(\dots)$

If object o has a field f , the field is accessed as $o.f$.

If object o has a method m , the method is invoked by $o.m(\dots)$.

If a class is the shape of a cookie (with its fields and methods), and objects are the cookies themselves, then $C(\dots)$ is a cookie-cutter that stamps out new cookies (with instances of C 's instance fields and methods).

In contrast, **class variables** are **unique**, and are **not** instantiated for each object. Rather, there is only one version of each class variable, and all objects of the class share access to it.



```
Object
 |
Pair
 |
Fraction
 |
Rational
```

Class definition: **Pair**

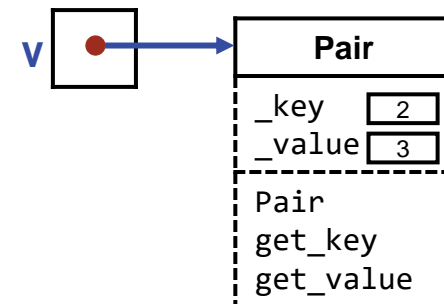
```
class Pair:
    # Representation
    _key: int
    _value: int

    # Constructor.
    def __init__(self, k: int, v: int) -> None:
        self._key = k; self._value = v

    # Access.
    def get_key(self) -> int: return self._key
    def get_value(self) -> int: return self._value
```

Variable declaration (with initialization):

```
v: Pair = Pair(2,3)
```



Object
|
Pair
|
Fraction
|
Rational

Execution of the variable declaration (with initialization) in four steps:

1. Create the variable v.

Class definition: Pair

```
class Pair:
    # Representation
    _key: int
    _value: int

    # Constructor.
    def __init__(self, k: int, v: int) -> None:
        self._key = k; self._value = v

    # Access.
    def get_key(self) -> int: return self._key
    def get_value(self) -> int: return self._value
```

Variable declaration (with initialization):

```
v: Pair = Pair(2,3)
```



```
Object
 |
Pair
 |
Fraction
 |
Rational
```

Execution of the variable declaration (with initialization) in four steps:

1. Create the variable `v`.
2. Create an object of type `Pair`.

Class definition: `Pair`

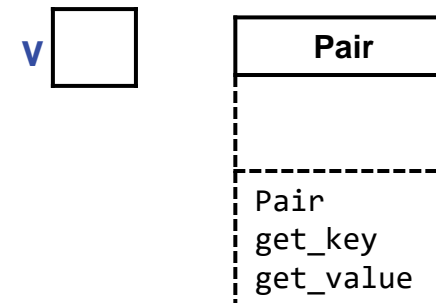
```
class Pair:
    # Representation
    _key: int
    _value: int

    # Constructor.
    def __init__(self, k: int, v: int) -> None:
        self._key = k; self._value = v

    # Access.
    def get_key(self) -> int: return self._key
    def get_value(self) -> int: return self._value
```

Variable declaration (with initialization):

```
v: Pair = Pair(2,3)
```




```

Object
 |
Pair
 |
Fraction
 |
Rational

```

Execution of the variable declaration (with initialization) in four steps:

1. Create the variable `v`.
2. Create an object of type `Pair`.
3. Invoke the constructor `__init__` on the object, which can establish fields and their values.

Class definition: `Pair`

```

class Pair:
    # Representation
    _key: int
    _value: int

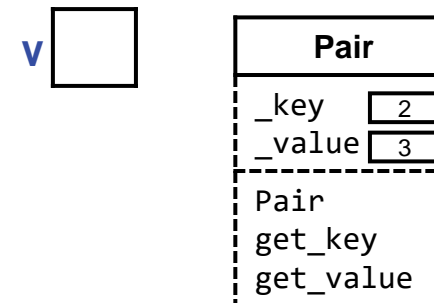
    # Constructor.
    def __init__(self, k: int, v: int) -> None:
        self._key = k; self._value = v

    # Access.
    def get_key(self) -> int: return self._key
    def get_value(self) -> int: return self._value

```

Variable declaration (with initialization):

```
v: Pair = Pair(2,3)
```



```

Object
 |
Pair
 |
Fraction
 |
Rational

```

Execution of the variable declaration (with initialization) in four steps:

1. Create the variable `v`.
2. Create an object of type `Pair`.
3. Invoke the constructor `__init__` on the object, which can establish fields and their values.
4. Assign a reference to the object in `v`.

Class definition: `Pair`

```

class Pair:
    # Representation
    _key: int
    _value: int

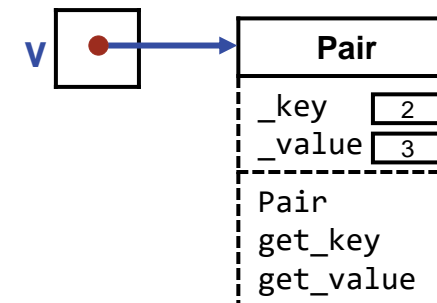
    # Constructor.
    def __init__(self, k: int, v: int) -> None:
        self._key = k; self._value = v

    # Access.
    def get_key(self) -> int: return self._key
    def get_value(self) -> int: return self._value

```

Variable declaration (with initialization):

```
v: Pair = Pair(2,3)
```



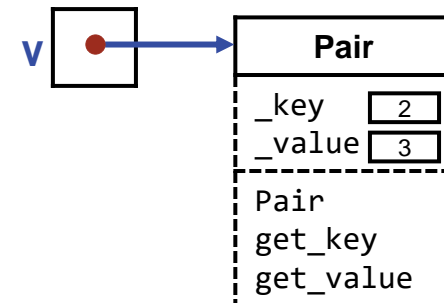
```
Object
 |
Pair
 |
Fraction
 |
Rational
```

Visibility: Each field and method of a class has visibility *public*, *private*, or *protected*.

```
class Pair:
    # Representation
    _key: int
    _value: int

    # Constructor.
    def __init__(self, k: int, v: int) -> None:
        self._key = k; self._value = v

    # Access.
    def get_key(self) -> int: return self._key
    def get_value(self) -> int: return self._value
```



Object
|
Pair
|
Fraction
|
Rational

A field or method name that does **not** begin with an underscore (“_” or “__”) is *public*.

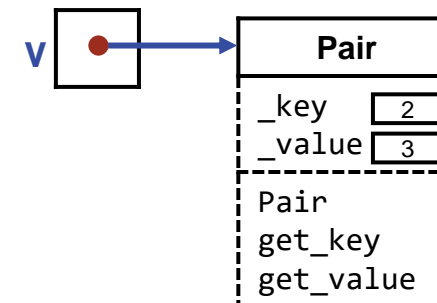
Visibility: Each field and method of a class has visibility *public*, *private*, or *protected*.

```
class Pair:
    # Representation
    _key: int
    _value: int

    # Constructor.
    def __init__(self, k: int, v: int) -> None:
        self._key = k; self._value = v

    # Access.
    def get_key(self) -> int: return self._key
    def get_value(self) -> int: return self._value
```

- *public* fields and methods are globally visible (the default).



```

Object
 |
Pair
 |
Fraction
 |
Rational

```

A field or method name that **begins** with double underscores (“__”) and doesn’t end with double underscores is *private*.

No *private* fields or method are illustrated

Visibility: Each field and method of a class has visibility *public*, *private*, or *protected*.

```

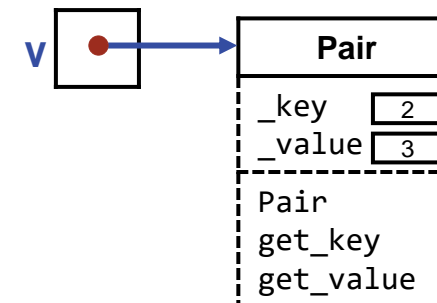
class Pair:
    # Representation
    _key: int
    _value: int

    # Constructor.
    def __init__(self, k: int, v: int) -> None:
        self._key = k; self._value = v

    # Access.
    def get_key(self) -> int: return self._key
    def get_value(self) -> int: return self._value

```

- *public* fields and methods are globally visible (the default).
- *private* fields and methods are only visible within the class.



```

Object
 |
Pair
 |
Fraction
 |
Rational

```

A field or method name that begins with a single underscore (“_”) is *protected* and should not be accessed outside of the class or its subclasses. Although such access it is *not* denied, the underscore serves as a *warning* to stay away.

Visibility: Each field and method of a class has visibility *public*, *private*, or *protected*.

```

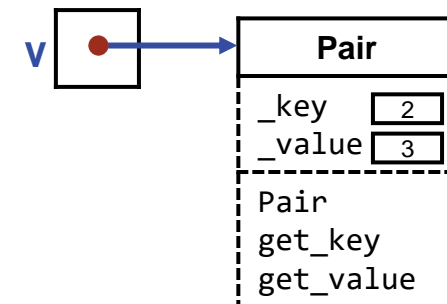
class Pair:
    # Representation
    _key: int
    _value: int

    # Constructor.
    def __init__(self, k: int, v: int) -> None:
        self._key = k; self._value = v

    # Access.
    def get_key(self) -> int: return self._key
    def get_value(self) -> int: return self._value

```

- *public* fields and methods are globally visible (the default).
- *private* fields and methods are only visible within the class.
- *protected* fields and methods are only visible within the class, or within a subclass of the class, e.g., Fraction.



```

Object
|
Pair
|
Fraction
|
Rational

```

Modifiability: class clients should consider *private* and *protected* fields as *read-only*.

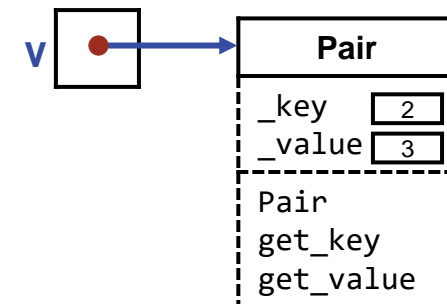
```

class Pair:
    # Representation
    _key: int
    _value: int

    # Constructor.
    def __init__(self, k: int, v: int) -> None:
        self._key = k; self._value = v

    # Access.
    def get_key(self) -> int: return self._key
    def get_value(self) -> int: return self._value

```



```

Object
|
Pair
|
Fraction
|
Rational

```

E.g., clients of `Pair` should obtain the values of `_key` and `_value` using the getters `get_key` and `get_value`, and should not access those fields directly. Such an object is said to be *immutable*.

Modifiability: class clients should consider *private* and *protected* fields as *read-only*.

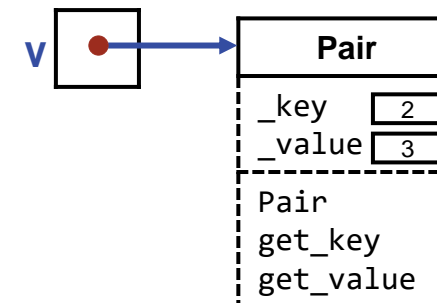
```

class Pair:
    # Representation
    _key: int
    _value: int

    # Constructor.
    def __init__(self, k: int, v: int) -> None:
        self._key = k; self._value = v

    # Access.
    def get_key(self) -> int: return self._key
    def get_value(self) -> int: return self._value

```




```

Object
 |
Pair
 |
Fraction
 |
Rational

```

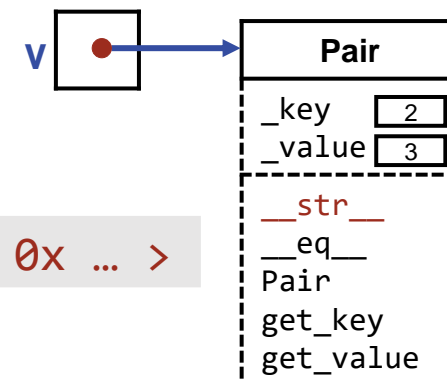
Default str representation:

- Every Pair is an **object**, and every **object** has a default `__str__` method.
- However, the str representation provided by that method is **not particularly helpful**.

Output the str representation of a Pair:

```
print(v)
```

```
<__main__.Pair object at 0x ... >
```



```
Object
|
Pair
|
Fraction
|
Rational
```

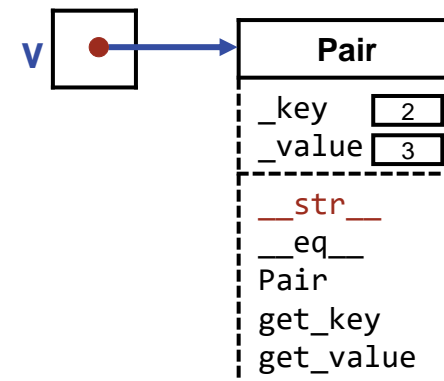
Overriding definition of `__str__` for Pair:

```
class Pair:
    ...
    # String representation.
    def __str__(self) -> str:
        return "<" + str(self._key) + "," + str(self._value) + ">"
```

Output the str representation of a Pair:

```
print(v)
```

```
<2,3>
```



Object
|
Pair
|
Fraction
|
Rational

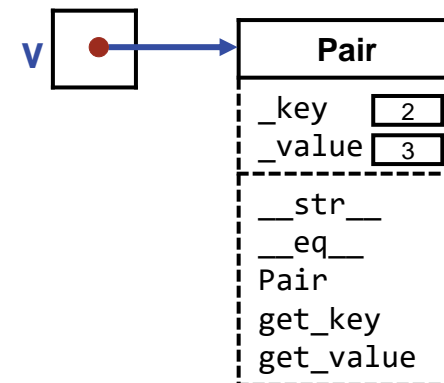
Execution of the print statement in three steps:

Overriding definition of `__str__` for Pair:

```
class Pair:
    ...
    # String representation.
    def __str__(self) -> str:
        return "<" + str(self._key) + "," + str(self._value) + ">"
```

Output the str representation of a Pair:

```
print(v)
```



```

Object
 |
Pair
 |
Fraction
 |
Rational

```

Execution of the print statement in three steps:

1. Obtain the value of variable v.

Overriding definition of `__str__` for Pair:

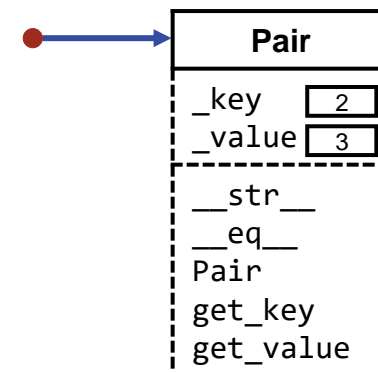
```

class Pair:
    ...
    # String representation.
    def __str__(self) -> str:
        return "<" + str(self._key) + "," + str(self._value) + ">"

```

Output the str representation of a Pair:

```
print(v)
```



```
Object
 |
Pair
 |
Fraction
 |
Rational
```

Execution of the print statement in three steps:

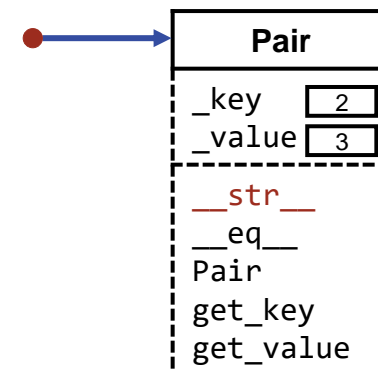
1. Obtain the value of variable v.
2. Compute the str representation of that value by invoking its `__str__` method.

Overriding definition of `__str__` for Pair:

```
class Pair:
    ...
    # String representation.
    def __str__(self) -> str:
        return "<" + str(self._key) + "," + str(self._value) + ">"
```

Output the str representation of a Pair:

```
print(v)
```



Object
|
Pair
|
Fraction
|
Rational

Execution of the print statement in three steps:

1. Obtain the value of variable v.
2. Compute the str representation of that value by invoking its `__str__` method.
3. Output that value.

Overriding definition of `__str__` for Pair:

```
class Pair:
    ...
    # String representation.
    def __str__(self) -> str:
        return "<" + str(self._key) + "," + str(self._value) + ">"
```

Output the str representation of a Pair:

```
print(v)
```

```
<2,3>
```

Object
|
Pair
|
Fraction
|
Rational

Execution of the print statement in three steps:

1. Obtain the value of variable `v`.
2. Compute the str representation of that value by invoking its `__str__` method.
3. Output that value.

Overriding definition of `__str__` for Pair:

```
class Pair:  
    ...  
    # String representation.  
    def __str__(self) -> str:  
        return "<" + str(self._key) + "," + str(self._value) + ">"
```

We have defined an overriding method `__str__` to provide a distinctive str representation for a `Pair`. Similarly, fixed-point `int` values (like `key` and `value`) need their str representations (as decimal numerals) in order that they may be concatenated with str values (like `"<"`, `","`, and `">"`). Function `str(...)` provides that representation.

Output the str representation of a Pair:

```
print(v)
```

```
<2,3>
```

Object
|
Pair
|
Fraction
|
Rational

Execution of the print statement in three steps:

1. Obtain the value of variable v.
2. Compute the str representation of that value by invoking its `__str__` method.
3. Output that value.

Overriding definition of `__str__` for Pair:

```
class Pair:  
    ...  
    # String representation.  
    def __str__(self) -> str:  
        return "<" + str(self._key) + "," + str(self._value) + ">"
```

You may wonder how in the output of the statement:

```
print("The decimal numeral for the int 2 is: ", 2)  
2 comes to be the decimal numeral for 2 without our needing to write:  
print("The decimal numeral for the int 2 is: ", str(2))  
but we won't go there.
```

Output the str representation of a Pair:

```
print(v)
```

```
<2,3>
```


Object
|
Pair
|
Fraction
|
Rational

The default definition of operator `==` for objects is identity.

- “Identity” means “exactly the same object”.

Demonstrate the difference between identity and equality.

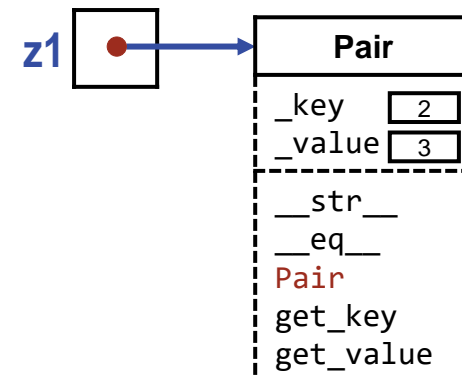
```
z1: Pair = Pair(2,3)
z2: Pair = Pair(2,3)
z3: Pair = z2
print(z1==z2)
print(z2==z3)
```

```
False
True
```

Object
|
Pair
|
Fraction
|
Rational

The default definition of operator `==` for objects is identity.

- “Identity” means “exactly the same object”.



Demonstrate the difference between identity and equality.

```
z1: Pair = Pair(2,3)
z2: Pair = Pair(2,3)
z3: Pair = z2
print(z1==z2)
print(z2==z3)
```

```

Object
 |
Pair
 |
Fraction
 |
Rational

```

The default definition of operator `==` for objects is identity.

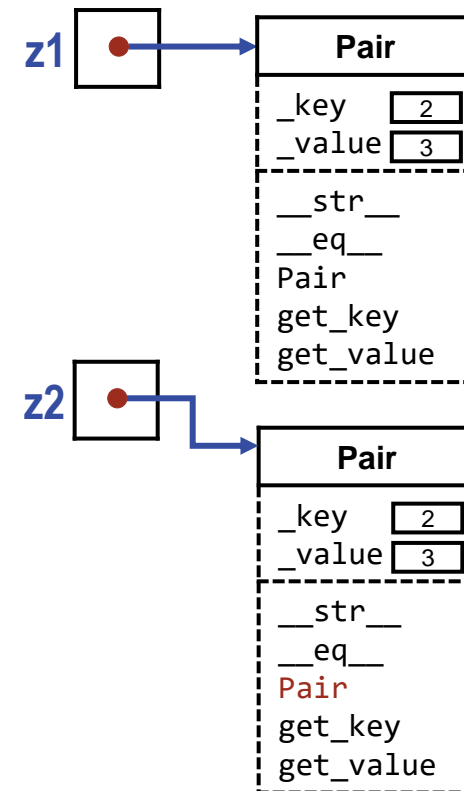
- “Identity” means “exactly the same object”.

Demonstrate the difference between identity and equality.

```

z1: Pair = Pair(2,3)
z2: Pair = Pair(2,3)
z3: Pair = z2
print(z1==z2)
print(z2==z3)

```



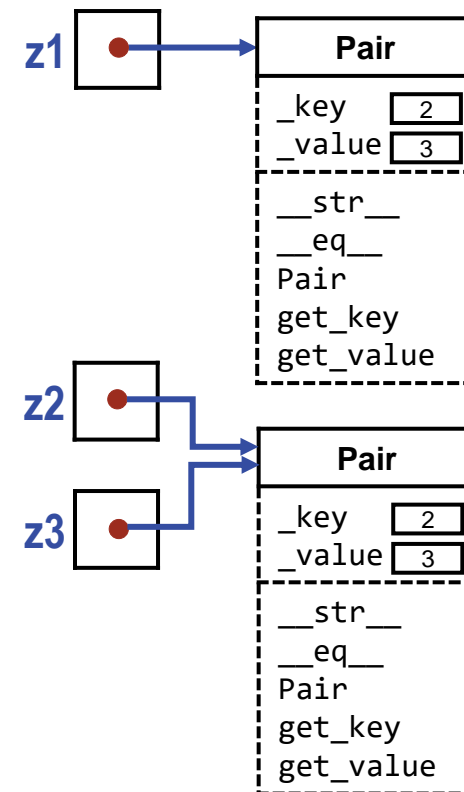
Object
|
Pair
|
Fraction
|
Rational

The default definition of operator `==` for objects is identity.

- “Identity” means “exactly the same object”.

Demonstrate the difference between identity and equality.

```
z1: Pair = Pair(2,3)
z2: Pair = Pair(2,3)
z3: Pair = z2
print(z1==z2)
print(z2==z3)
```



Object
|
Pair
|
Fraction
|
Rational

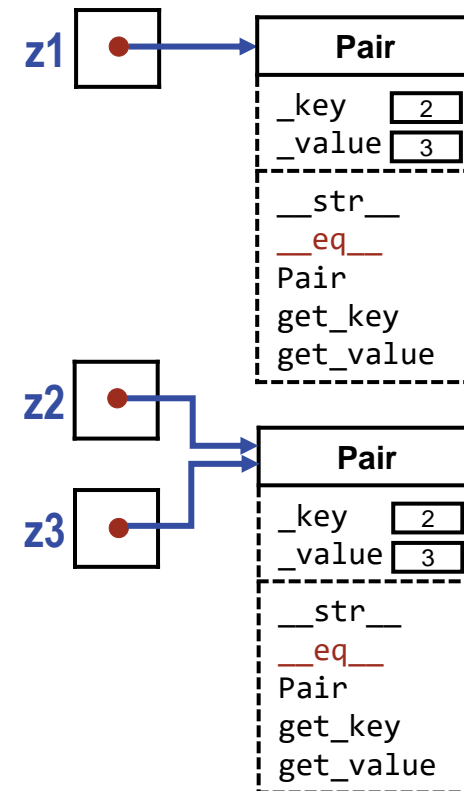
The default definition of operator `==` for objects is identity.

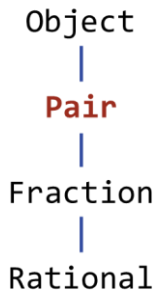
- “Identity” means “exactly the same object”.

Demonstrate the difference between identity and equality.

```
z1: Pair = Pair(2,3)
z2: Pair = Pair(2,3)
z3: Pair = z2
print(z1==z2)
print(z2==z3)
```

False





The default definition of operator == for objects is identity.

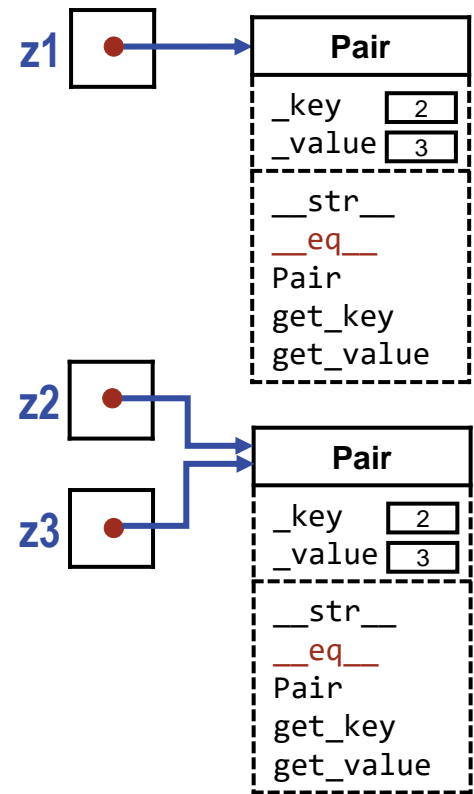
- “Identity” means “exactly the same object”.

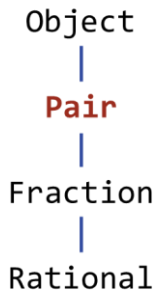
Demonstrate the difference between identity and equality.

```

z1: Pair = Pair(2,3)
z2: Pair = Pair(2,3)
z3: Pair = z2
print(z1==z2)
print(z2==z3)
  
```

False
True





The default definition of `__eq__` for object values is identity.

- “Identity” means “exactly the same object”.
- Every **object** has an `__eq__` method that is used to test “equality” of two **objects**.
- The default definition of method `__eq__` for any object is identity, i.e., this is where the default behavior of `==` came from.

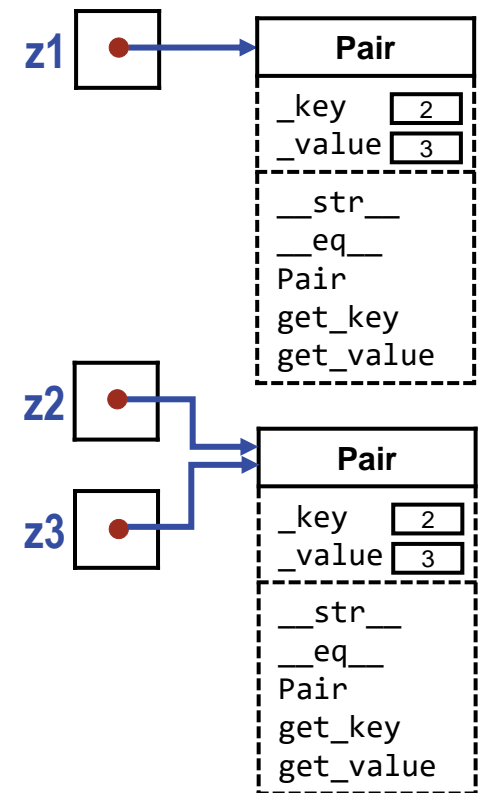
Demonstrate the difference between identity and equality.

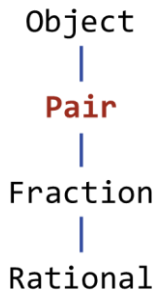
```

z1: Pair = Pair(2,3)
z2: Pair = Pair(2,3)
z3: Pair = z2
print(z1==z2)
print(z2==z3)
  
```

with the default definition of `__eq__`, i.e., identity

False
True





The default definition of `__eq__` can be overridden.

- “Identity” means “exactly the same object”.
- Every **object** has an `__eq__` method that is used to test “equality” of two **objects**.
- The default definition of method `__eq__` for any object is identity, i.e., this is where the default behavior of `==` came from.
- The definition of `__eq__` can be overridden, e.g., to treat non-identical pairs with equal components as equal.

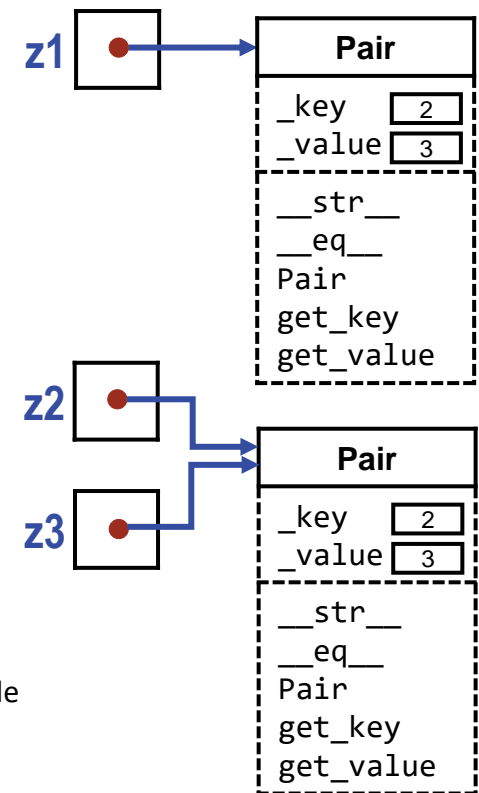
Demonstrate the difference between identity and equality.

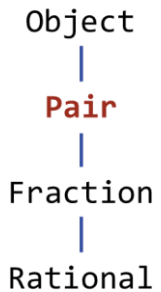
```

z1: Pair = Pair(2,3)
z2: Pair = Pair(2,3)
z3: Pair = z2
print(z1==z2)
print(z2==z3)
  
```

with the **overriding definition** of `__eq__` shown on the next slide

True
True





Demonstrate the difference between identity and equality.

```

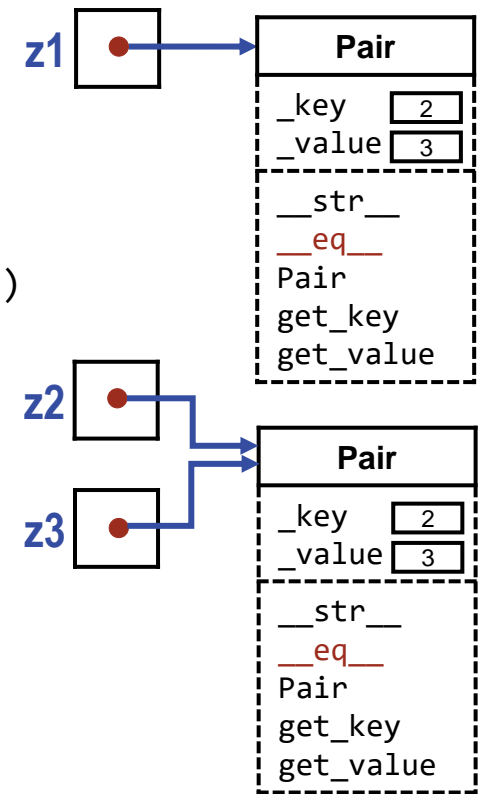
z1: Pair = Pair(2,3)
z2: Pair = Pair(2,3)
z3: Pair = z2
print(z1==z2)
print(z2==z3)
  
```

True
True

Overriding definition of `__eq__` of Pair.

```

class Pair:
    ...
    # Equality.
    def __eq__(self, other) -> bool:
        if other is None: return False
        if other is self: return True
        if not isinstance(other, Pair): return False;
        return (self._key == other._key) and (self._value == other._value)
  
```



Object
|
Pair
|
Fraction
|
Rational

Operator "is" tests identity.

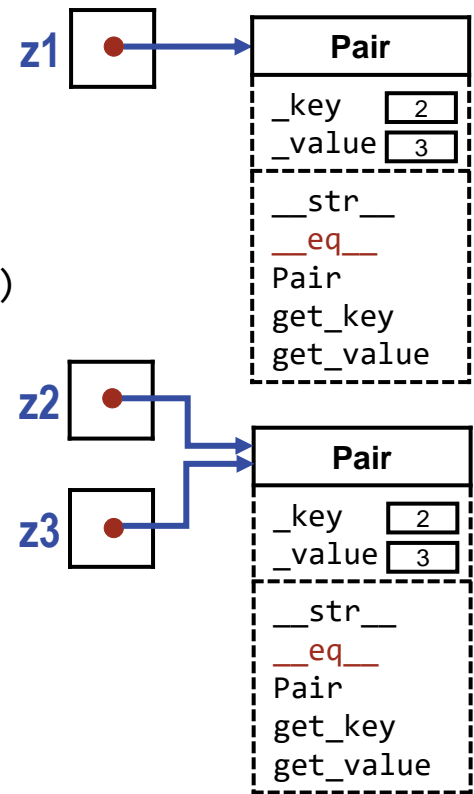
Demonstrate the difference between identity and equality.

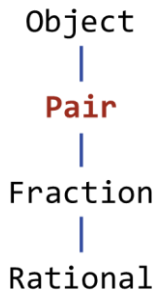
```
z1: Pair = Pair(2,3)
z2: Pair = Pair(2,3)
z3: Pair = z2
print(z1==z2)
print(z2==z3)
```

True
True

Overriding definition of `__eq__` of Pair.

```
class Pair:
    ...
    # Equality.
    def __eq__(self, other) -> bool:
        if other is None: return False
        if other is self: return True
        if not isinstance(other, Pair): return False;
        return (self._key == other._key) and (self._value == other._value)
```





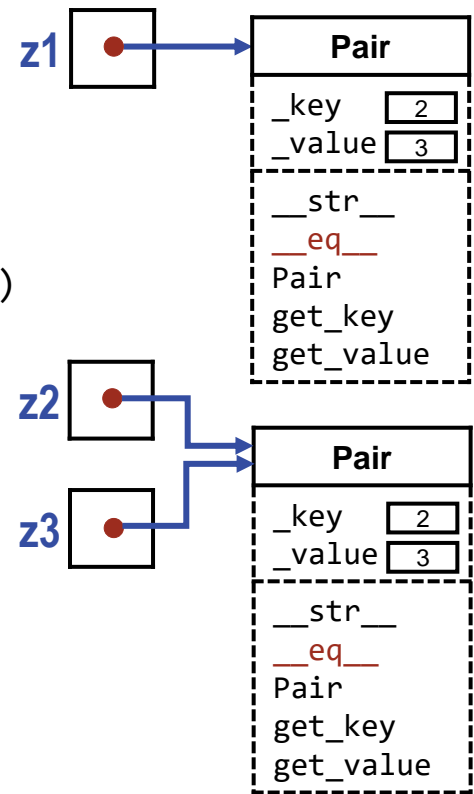
Demonstrate the difference between identity and equality.

```
z1: Pair = Pair(2,3)  
z2: Pair = Pair(2,3)  
z3: Pair = z2  
print(z1==z2)  
print(z2==z3)
```

True
True

Overriding definition of `__eq__` of Pair.

```
class Pair:  
    ...  
    # Equality.  
    def __eq__(self, other) -> bool:  
        if other is None: return False  
        if other is self: return True  
        if not isinstance(other, Pair): return False;  
        return (self._key == other._key) and (self._value == other._value)
```



An object is never equal to no object.

```
Object
|
Pair
|
Fraction
|
Rational
```

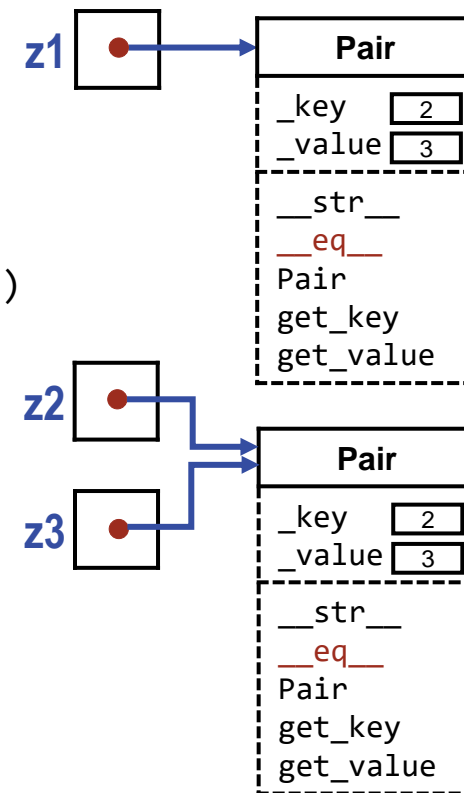
Demonstrate the difference between identity and equality.

```
z1: Pair = Pair(2,3)
z2: Pair = Pair(2,3)
z3: Pair = z2
print(z1==z2)
print(z2==z3)
```

```
True
True
```

Overriding definition of `__eq__` of Pair.

```
class Pair:
    ...
    # Equality.
    def __eq__(self, other) -> bool:
        if other is None: return False
        if other is self: return True
        if not isinstance(other, Pair): return False;
        return (self._key == other._key) and (self._value == other._value)
```



An object is always equal to itself, e.g. z2 and z3.

```
Object
|
Pair
|
Fraction
|
Rational
```

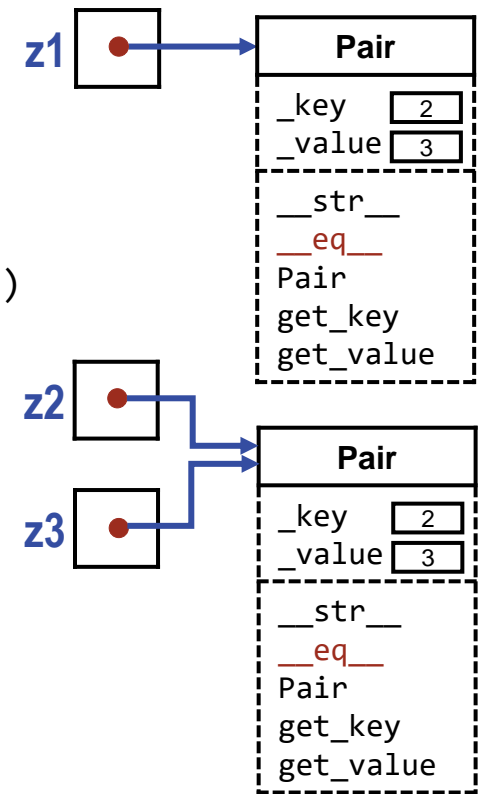
Demonstrate the difference between identity and equality.

```
z1: Pair = Pair(2,3)
z2: Pair = Pair(2,3)
z3: Pair = z2
print(z1==z2)
print(z2==z3)
```

```
True
True
```

Overriding definition of `__eq__` of Pair.

```
class Pair:
    ...
    # Equality.
    def __eq__(self, other) -> bool:
        if other is None: return False
        if other is self: return True
        if not isinstance(other, Pair): return False;
        return (self._key == other._key) and (self._value == other._value)
```



A Pair can only equal another Pair.

Object
|
Pair
|
Fraction
|
Rational

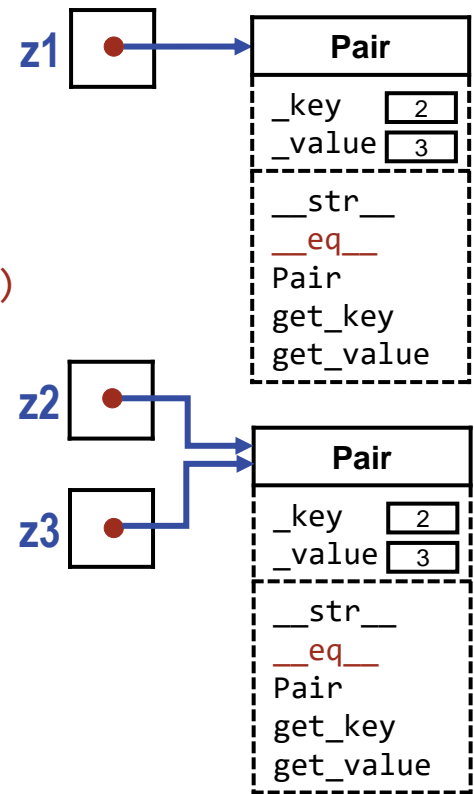
Demonstrate the difference between identity and equality.

```
z1: Pair = Pair(2,3)
z2: Pair = Pair(2,3)
z3: Pair = z2
print(z1==z2)
print(z2==z3)
```

True
True

Overriding definition of `__eq__` of Pair.

```
class Pair:
    ...
    # Equality.
    def __eq__(self, other) -> bool:
        if other is None: return False
        if other is self: return True
        if not isinstance(other, Pair): return False;
        return (self._key == other._key) and (self._value == other._value)
```



A Pair can only equal another Pair, and then only when their components are equal, e.g. z1 and z2.

```
Object
 |
Pair
 |
Fraction
 |
Rational
```

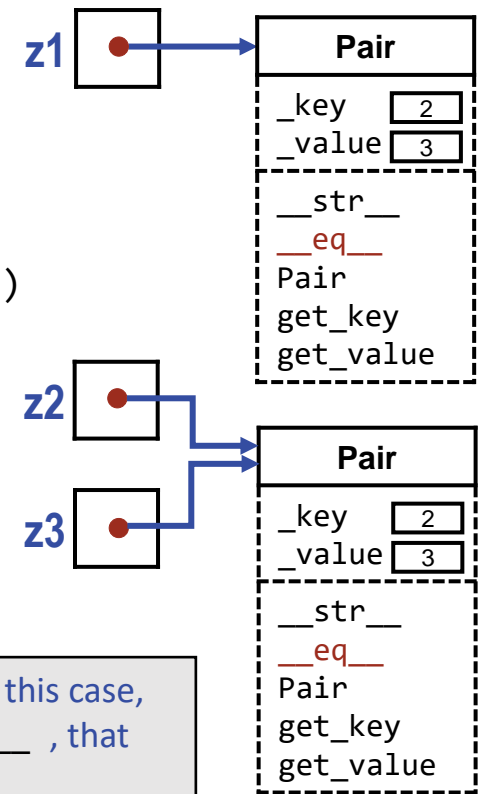
Demonstrate the difference between identity and equality.

```
z1: Pair = Pair(2,3)
z2: Pair = Pair(2,3)
z3: Pair = z2
print(z1==z2)
print(z2==z3)
```

```
True
True
```

Overriding definition of `__eq__` of Pair.

```
class Pair:
    ...
    # Equality.
    def __eq__(self, other) -> bool:
        if other is None: return False
        if other is self: return True
        if not isinstance(other, Pair): return False;
        return (self._key == other._key) and (self._value == other._value)
```



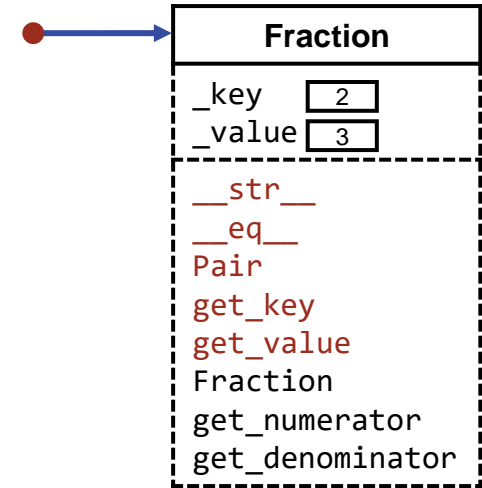
N.B. The left and right operands of `==` may be of different types, each with their own definition of `__eq__`. In this case, you may legitimately ask which `__eq__` is used? If only one operand type has an overriding definition of `__eq__`, that operand's definition is used, otherwise preference is given to the left operand's definition

Object
|
Pair
|
Fraction
|
Rational

Fraction is a subclass of Pair, and as such acquires the fields and methods from Pair.

Subclass definition: Fraction

```
class Fraction(Pair):  
    # Constructor.  
    def __init__(self, numerator: int, denominator: int) -> Fraction:  
        super(numerator, denominator); # Apply the Pair constructor.  
        assert denominator != 0, "0 denominator"  
  
    # Access.  
    def get_numerator(self) -> int: return self._key  
    def get_denominator(self) -> int: return self._value
```

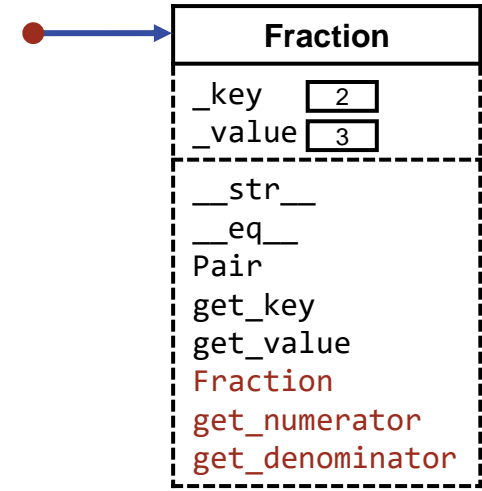


Object
|
Pair
|
Fraction
|
Rational

Fraction is a subclass of Pair, and as such acquires the fields and methods from Pair, while adding more of its own.

Subclass definition: Fraction

```
class Fraction(Pair):  
    # Constructor.  
    def __init__(self, numerator: int, denominator: int) -> Fraction:  
        super(numerator, denominator); # Apply the Pair constructor.  
        assert denominator != 0, "0 denominator"  
  
    # Access.  
    def get_numerator(self) -> int: return self._key  
    def get_denominator(self) -> int: return self._value
```



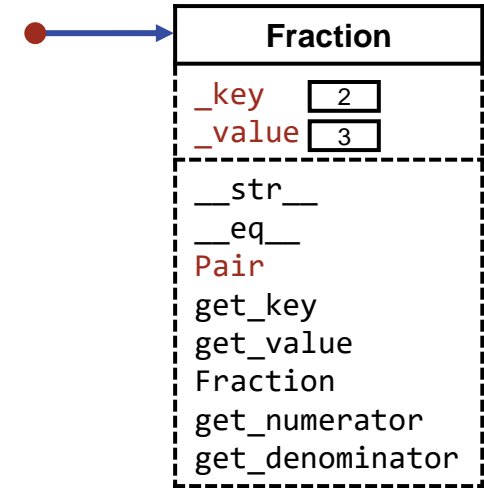


Subclass definition: Fraction

```

class Fraction(Pair):
    # Constructor.
    def __init__(self, numerator: int, denominator: int) -> Fraction:
        super(numerator, denominator); # Apply the Pair constructor.
        assert denominator != 0, "0 denominator"

    # Access.
    def get_numerator(self) -> int: return self._key
    def get_denominator(self) -> int: return self._value
  
```



The Fraction constructor uses the Pair constructor to set fields `_key` and `_value`.

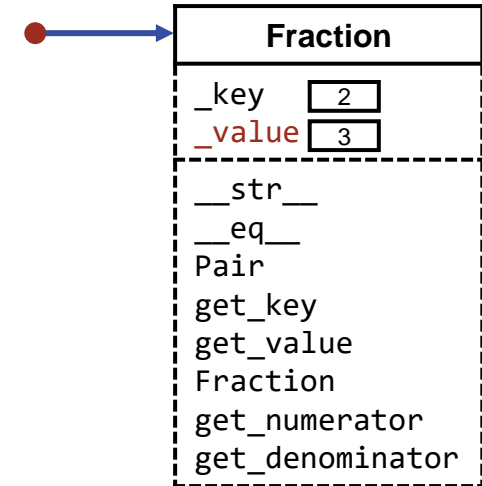


Subclass definition: Fraction

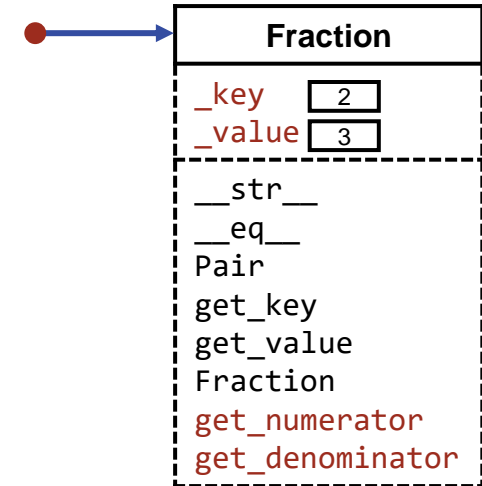
```

class Fraction(Pair):
    # Constructor.
    def __init__(self, numerator: int, denominator: int) -> Fraction:
        super(numerator, denominator); # Apply the Pair constructor.
        assert denominator != 0, "0 denominator"

    # Access.
    def get_numerator(self) -> int: return self._key
    def get_denominator(self) -> int: return self._value
  
```



It then assures that the denominator is not zero.



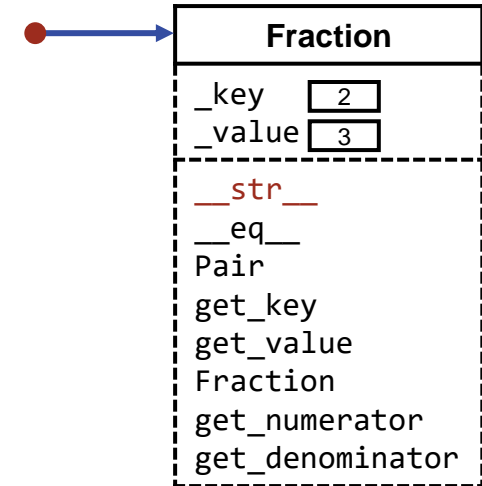
Subclass definition: Fraction

```

class Fraction(Pair):
    # Constructor.
    def __init__(self, numerator: int, denominator: int) -> None:
        super(numerator, denominator); # Apply the Pair constructor.
        assert denominator != 0, "0 denominator"

    # Access.
    def get_numerator(self) -> int: return self._key
    def get_denominator(self) -> int: return self._value
  
```

The getters have direct access to the fields `_key` and `_value` because they are *protected* in `Pair`, a superclass of `Fraction`.



Overriding definition of `__str__` for Fraction:

```

class Fraction(Pair):
    # Constructor.
    def __init__(self, numerator: int, denominator: int) -> None:
        super(numerator, denominator); # Apply the Pair constructor.
        assert denominator != 0, "0 denominator"

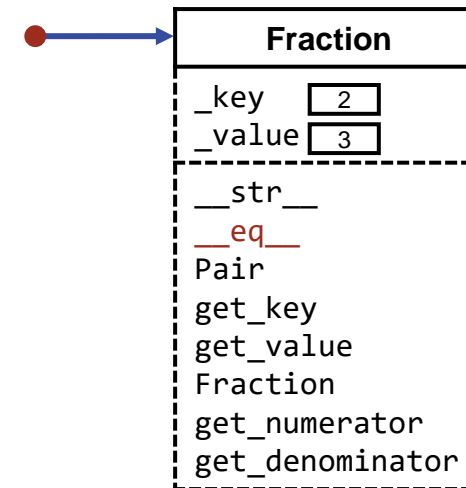
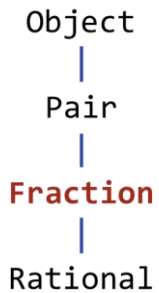
    # Access.
    def get_numerator(self) -> int: return self._key
    def get_denominator(self) -> int: return self._value

    # String representation.
    def __str__(self) -> str: return str(self._key) + "/" + str(self._value)
  
```

Different string representations for Pair and Fraction

```
print(Pair(2,3), Fraction(2,3))
```

```
<2,3> 2/3
```



The definition of `__eq__` for Fraction:

Two fractions are equal iff they have equal numerators and equal denominators. This is (almost) the test that is used to test the equality of two `Pairs`, so we might consider omitting an overriding definition of `__eq__` for `Fraction`, and rely on the definition in `Pair`.

However, pairs and fractions are two fundamentally different sorts of things, and it seems inappropriate to let a `Fraction` be considered equal to a `Pair` just because they happen to have the same two equal fields.

The effect of Fraction relying on the definition of `__eq__` in Pair

```

z1: Pair = Pair(2,3)
z2: Fraction = Fraction(2,3)
print(z1, z2)
print(z1==z2)
  
```

```

<2,3> 2/3
True
  
```

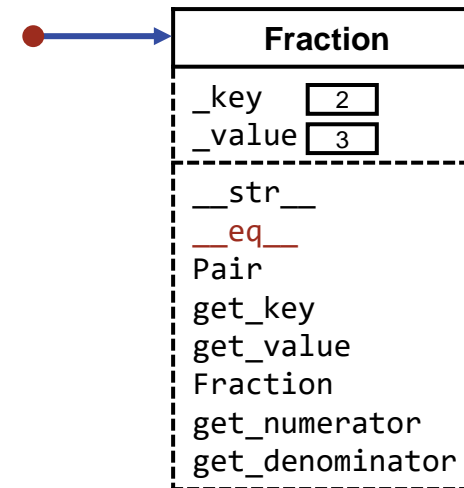


The overriding definition of `__eq__` for Fraction:

Accordingly, we choose to give `Fraction` its own definition of `__eq__`, and so treat fractions as fundamentally different from pairs.

```

class Fraction(Pair):
    ...
    # Equality.
    def __eq__(self, other) -> bool:
        if other is None: return False
        if other is self: return True
        if not isinstance(other, Fraction): return False;
        return (self._key == other._key) and (self._value == other._value)
  
```



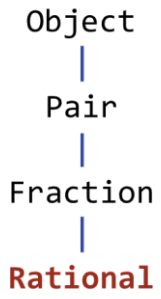
The effect of Fraction getting its own definition of `__eq__`

```

z1: Pair = Pair(2,3)
z2: Fraction = Fraction(2,3)
print(z1, z2)
print(z1==z2)
  
```

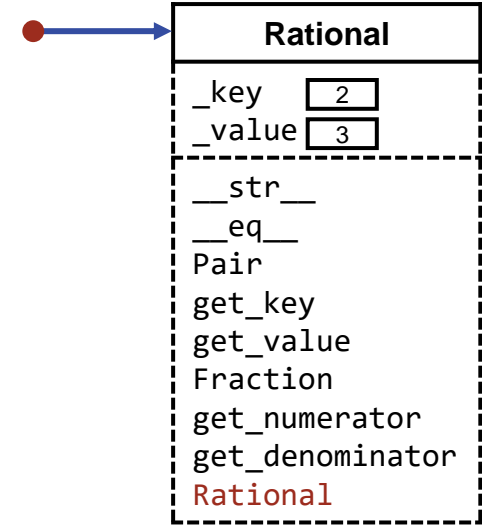
```

<2,3> 2/3
False
  
```



Subclass definition: Rational

```
class Rational(Fraction):
```

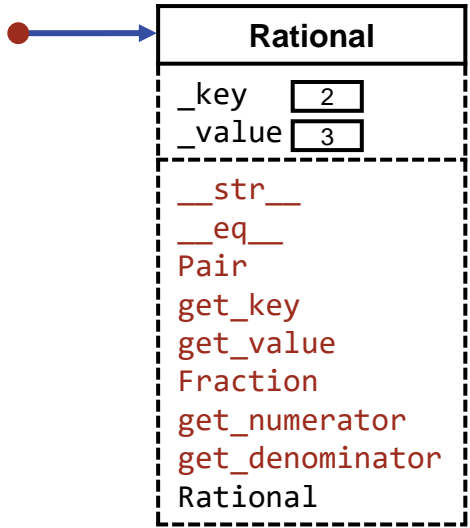


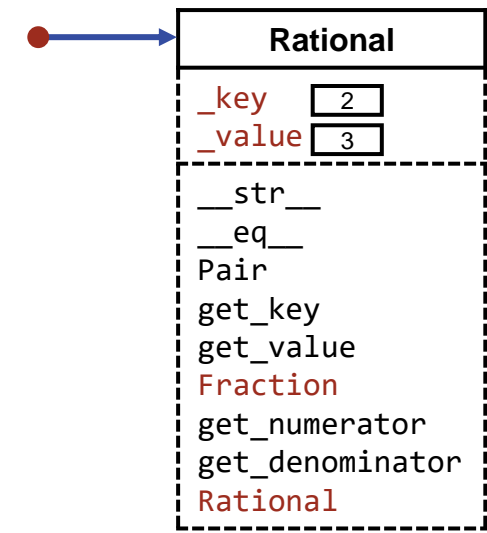
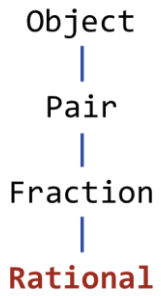
Object
|
Pair
|
Fraction
|
Rational

Rational is a subclass of Fraction, and as such acquires fields and methods of a Fraction.

Subclass definition: Rational

```
class Rational(Fraction):
```





Subclass definition: Rational

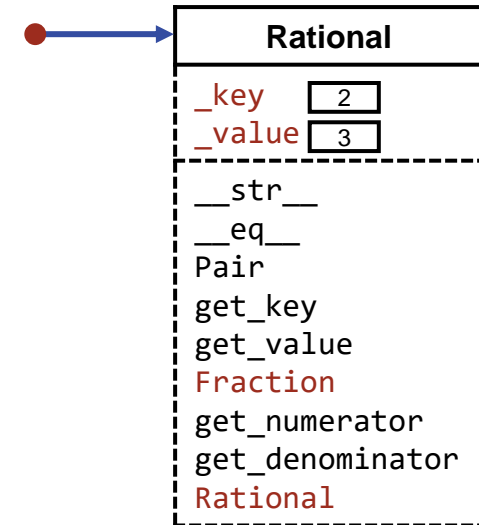
```
class Rational(Fraction):  
    # Constructor.  
    def __init__(numerator: int, denominator: int):  
        super().__init__(numerator, denominator)    # Apply the Fraction constructor.
```

The Rational constructor uses the Fraction constructor to set fields `_key` and `_value`, and to check that the denominator is not zero.

```

Object
 |
Pair
 |
Fraction
 |
Rational

```



Subclass definition: Rational

```

class Rational(Fraction):
    # Constructor.
    def __init__(numerator: int, denominator: int):
        super().__init__(numerator, denominator) # Apply the Fraction constructor.
        g: int = gcd(numerator, denominator)
        self._key = numerator // g
        self._value = denominator // g

```

The Rational constructor uses the Fraction constructor to set fields `_key` and `_value`, and to check that the denominator is not zero. Then it updates the representation to reduced form, i.e., no common factors.

```

Object
 |
Pair
 |
Fraction
 |
Rational

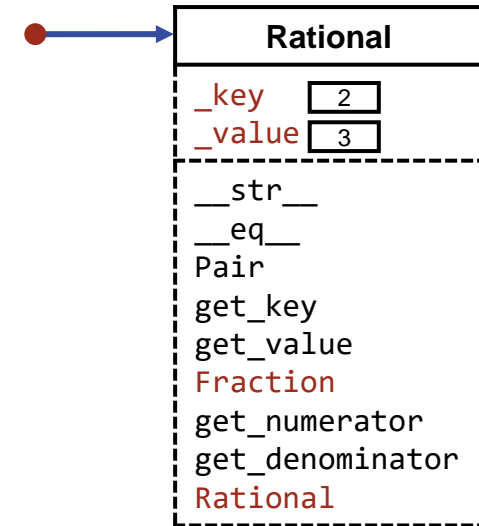
```

Subclass definition: Rational

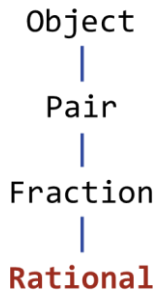
```

class Rational(Fraction):
    # Constructor.
    def __init__(numerator: int, denominator: int):
        super().__init__(numerator, denominator) # Apply the Fraction constructor.
        g: int = gcd(numerator, denominator)
        self._key = numerator // g
        self._value = denominator // g

```

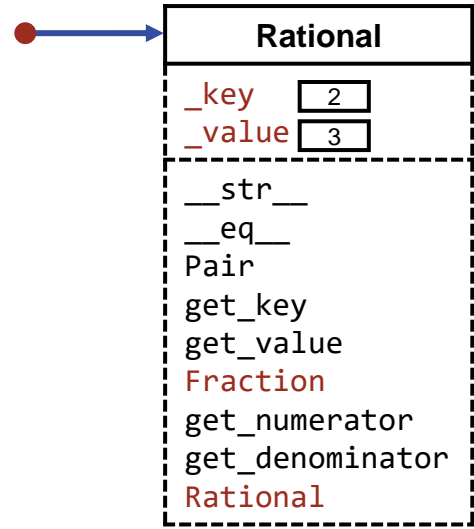


 **Boundary conditions. Dead last, but don't forget them.**



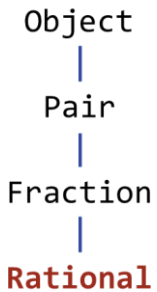
Subclass definition: Rational

```
class Rational(Fraction):  
    # Constructor.  
    def __init__(numerator: int, denominator: int):  
        super().__init__(numerator, denominator) # Apply the Fraction constructor.  
        g: int = gcd(numerator, denominator)  
        self._key = numerator // g  
        self._value = denominator // g
```



Function gcd will fail for negative arguments!

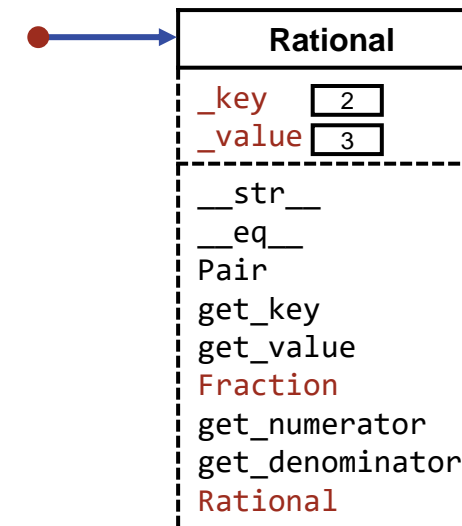
Boundary conditions. Dead last, but don't forget them.



Subclass definition: Rational

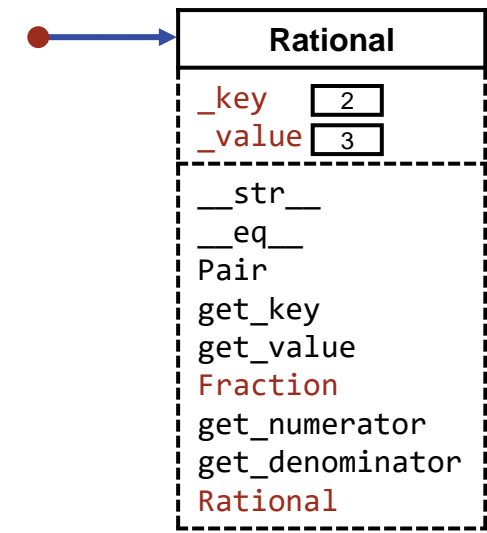
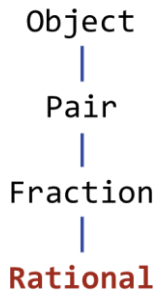
```

class Rational(Fraction):
    # Constructor.
    def __init__(numerator: int, denominator: int):
        super().__init__(numerator, denominator) # Apply the Fraction constructor.
        g: int = gcd(numerator, denominator)
        self._key = numerator // g
        self._value = denominator // g
  
```



Function gcd will fail for non-positive arguments!

 **Boundary conditions. Dead last, but don't forget them.**



Subclass definition: Rational

```

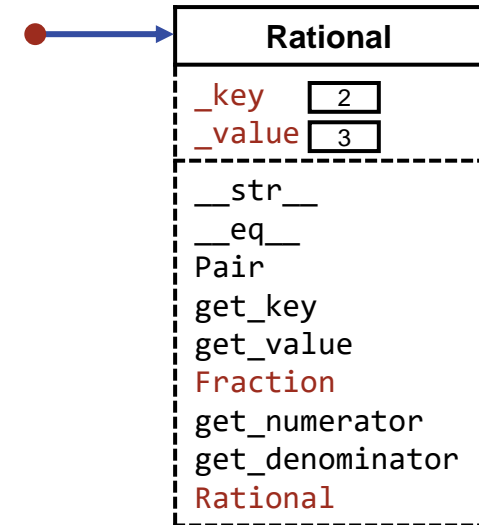
class Rational(Fraction):
    # Constructor.
    def __init__(numerator: int, denominator: int):
        super().__init__(numerator, denominator) # Apply the Fraction constructor.
        g: int = gcd(numerator, denominator)
        self._key = numerator // g
        self._value = denominator // g
  
```

Reduced form is good, i.e., no common factors, but canonical form is better.

```

Object
 |
Pair
 |
Fraction
 |
Rational

```



Subclass definition: Rational

```

class Rational(Fraction):
    # Constructor.
    def __init__(numerator: int, denominator: int):
        super().__init__(numerator, denominator) # Apply the Fraction constructor.
        g: int = gcd(numerator, denominator)
        self._key = numerator // g
        self._value = denominator // g

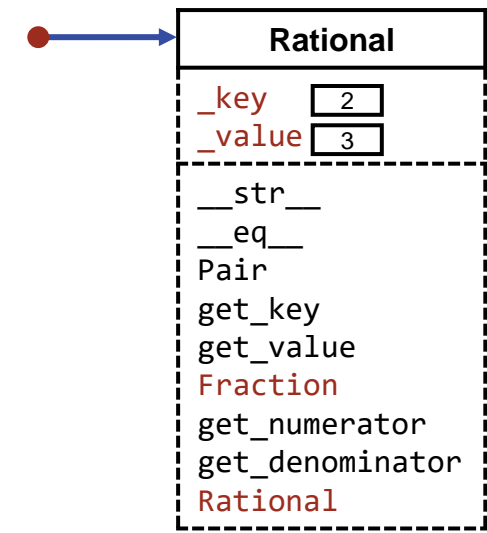
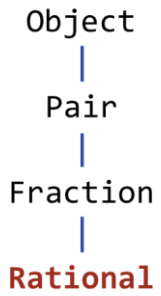
```

Reduced form is good, i.e., no common factors, but canonical form is better. Equal rationals should have the same representations:

- Zero should have a denominator of 1.
- Negatives should have a negative numerator and a positive denominator.
- Positives should have positive numerator and denominator.



Boundary conditions. Dead last, but don't forget them.



Subclass definition: Rational

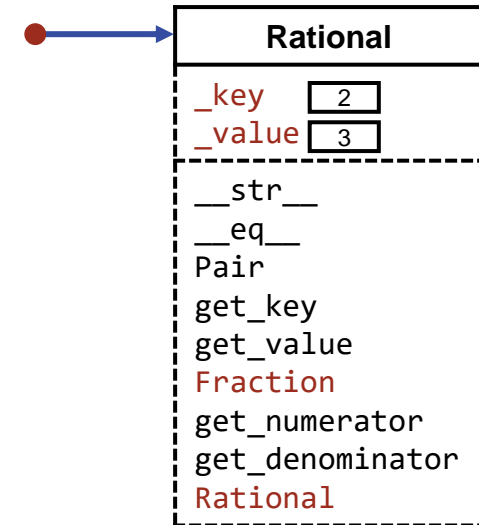
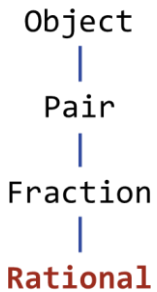
```

class Rational(Fraction):
    # Constructor.
    def __init__(numerator: int, denominator: int):
        super().__init__(numerator, denominator) # Apply the Fraction constructor.
        if numerator == 0: self._value = 1
  
```

Reduced form is good, i.e., no common factors, but canonical form is better. Equal rationals should have the same representations:

- Zero should have a denominator of 1.
-

Boundary conditions. Dead last, but don't forget them.



Subclass definition: Rational

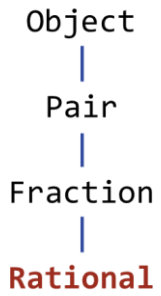
```

class Rational(Fraction):
    # Constructor.
    def __init__(numerator: int, denominator: int):
        super().__init__(numerator, denominator) # Apply the Fraction constructor.
        if numerator == 0: self._value = 1
        else:
            g: int = gcd(abs(numerator), abs(denominator))
            if ((numerator < 0) and (denominator > 0)) or (
                (numerator > 0) and (denominator < 0) ): sign = -1
            else:
                sign = +1
            self._key = sign * abs(numerator) // g
            self._value = abs(denominator) // g
  
```

Reduced form is good, i.e., no common factors, but canonical form is better. Equal rationals should have the same representations:

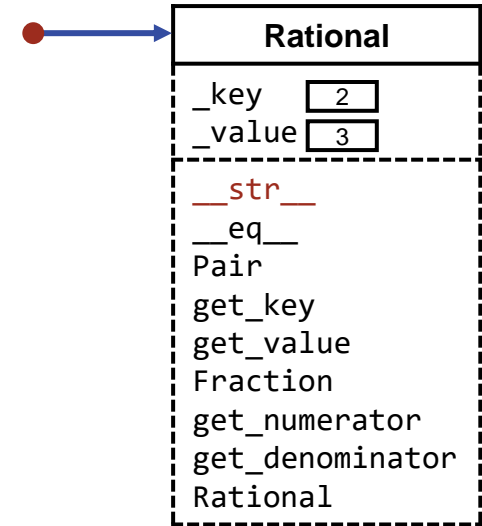
- Zero should have a denominator of 1.
- Negatives should have a negative numerator and a positive denominator.
- Positives should have positive numerator and denominator.

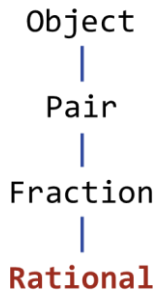
Boundary conditions. Dead last, but don't forget them.



Overriding definition of `__str__` for Rational:

```
class Rational(Fraction):  
    ...  
    # String representation.  
    def __str__(self) -> str:  
        if self._value == 1: return str(self._key) # self as an integer  
        else: return super().__str__() # self as a Fraction
```





Overriding definition of `__eq__` for `Rational` is not needed.

Rationals are fractions in canonical form, and are equal iff they have equal numerators and equal denominators. We choose to consider a fraction that is serendipitously in canonical form as equal to a rational with the same numerator and denominator. We choose to consider a fraction that is not in canonical form as unequal to the rational which is that fraction in canonical form.

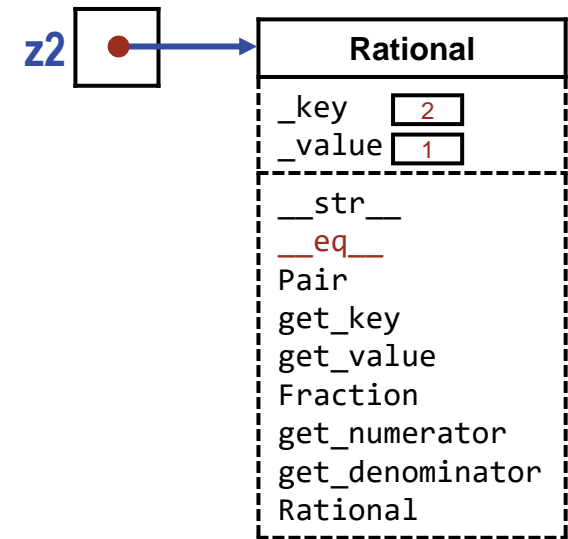
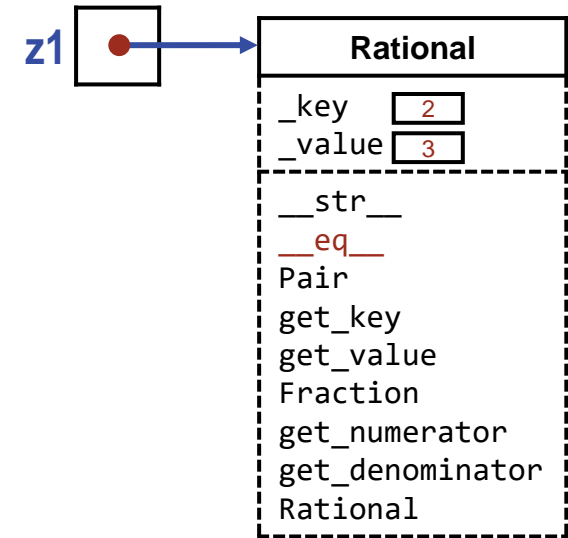
The effect of letting `Rational` rely on the definition of `__eq__` in `Fraction`

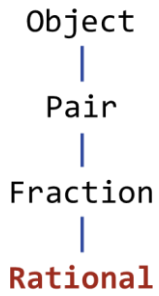
```

Rational z1 = Rational(4,6)
Rational z2 = Rational(6,3)
print(z1, z2)
print(z1 == z2)
  
```

```

2/3 2
false
  
```





Overriding definition of `__eq__` for `Rational` is not needed.

Rationals are fractions in canonical form, and are equal iff they have equal numerators and equal denominators. **We choose to consider a fraction that is serendipitously in canonical form as equal to a rational with the same numerator and denominator.** We choose to consider a fraction that is not in canonical form as unequal to the rational which is that fraction in canonical form.

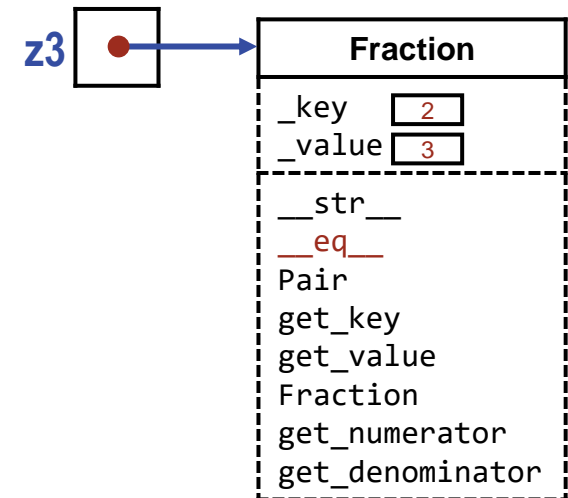
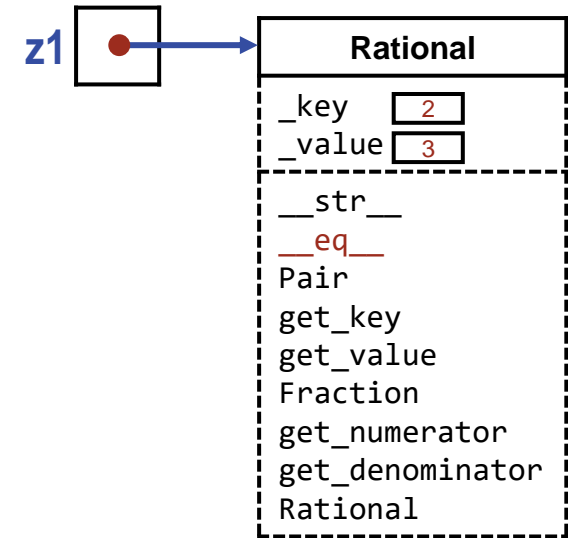
The effect of letting `Rational` rely on the definition of `__eq__` in `Fraction`

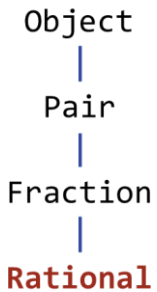
```

Rational z1 = Rational(4,6)
Fraction z3 = Fraction(2,3)
print(z1, z3)
print(z1 == z3)
  
```

```

2/3 2/3
true
  
```





Overriding definition of `__eq__` for `Rational` is not needed.

Rationals are fractions in canonical form, and are equal iff they have equal numerators and equal denominators. We choose to consider a fraction that is serendipitously in canonical form as equal to a rational with the same numerator and denominator. **We choose to consider a fraction that is not in canonical form as unequal to the rational which is that fraction in canonical form.**

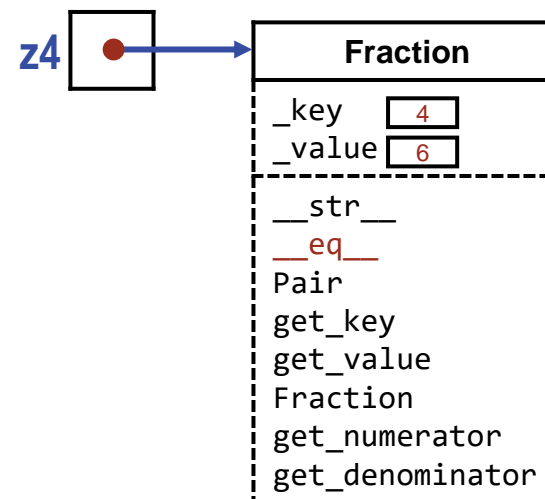
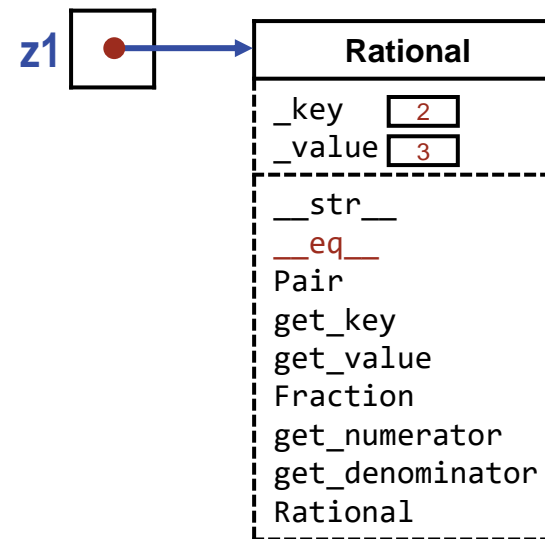
The effect of letting `Rational` rely on the definition of `__eq__` in `Fraction`

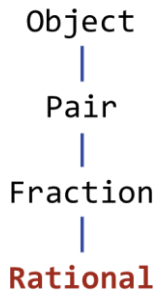
```

Rational z1 = Rational(4,6)
Fraction z4 = Fraction(4,6)
print(z1, z4)
print(z1 == z4)
  
```

```

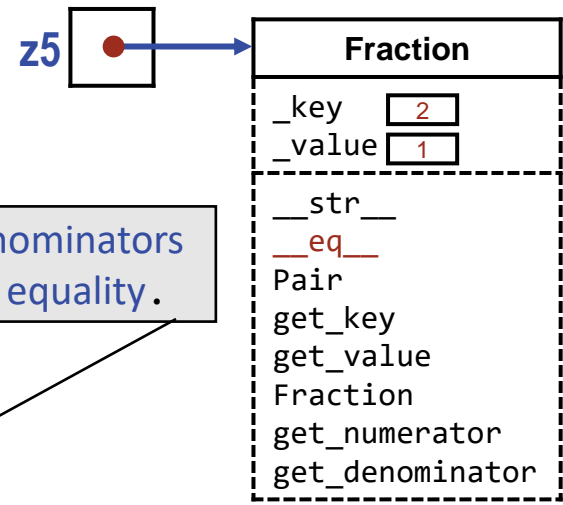
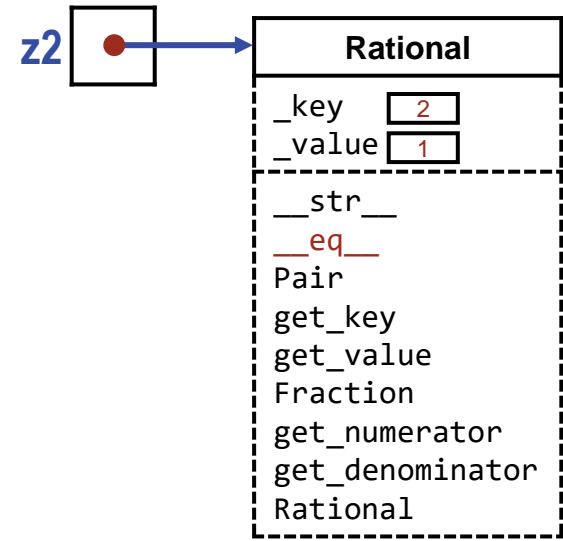
2/3 4/6
false
  
```





Overriding definition of `__eq__` for `Rational` is not needed.

Rationals are fractions in canonical form, and are equal iff they have equal numerators and equal denominators. We choose to consider a fraction that is serendipitously in canonical form as equal to a rational with the same numerator and denominator. We choose to consider a fraction that is not in canonical form as unequal to the rational which is that fraction in canonical form.



The display of rationals with denominators of 1 as integers has no effect on equality.

The effect of letting `Rational` rely on the definition of `__eq__` in `Fraction`

```
Rational z2 = Rational(6,3)
Fraction z5 = Fraction(2,1)
print(z2, z5)
print(z2 == z5)
```

```
2 2/1
true
```

Unit Test: Cover **every public aspect** of the class's interface (*black-box* testing), and if you know the implementation internals, **every corner case** you can foresee (*white-box* testing).

Test code	Output
<code>print(Rational(2,3))</code>	2/3
<code>print(Rational(4,6))</code>	2/3
<code>print(Rational(-4,6))</code>	-2/3
<code>print(Rational(4,-6))</code>	-2/3
<code>print(Rational(-4,-6))</code>	2/3
<code>print(Rational(6,3))</code>	2
<code>print(Rational(0,1))</code>	0
<code>print(Rational(0,10))</code>	0
<code>print(Rational(0,-10))</code>	0
<code>print(Rational(2,3) == Rational(4,6))</code>	True

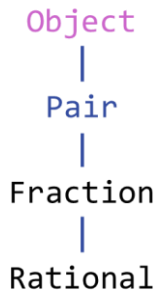
Can you think of other examples to test?

Object
|
Pair
|
Fraction
|
Rational

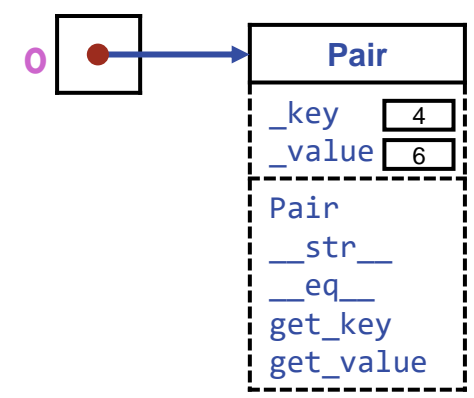


Subtype polymorphism: A variable of class C can be assigned a reference to any object of class C', where C' is either C itself, or C' is a subclass of C, i.e., lower in the class hierarchy.

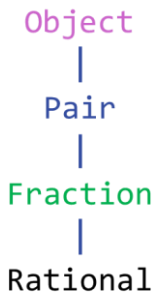
o: object



Subtype polymorphism:

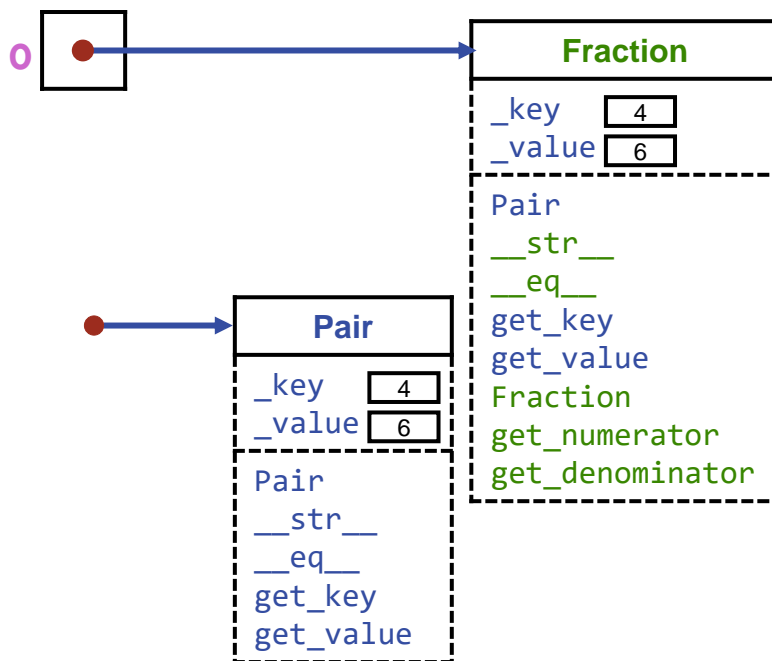


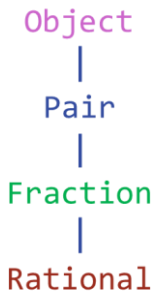
```
o: object  
o = Pair(4,6)
```



Subtype polymorphism:

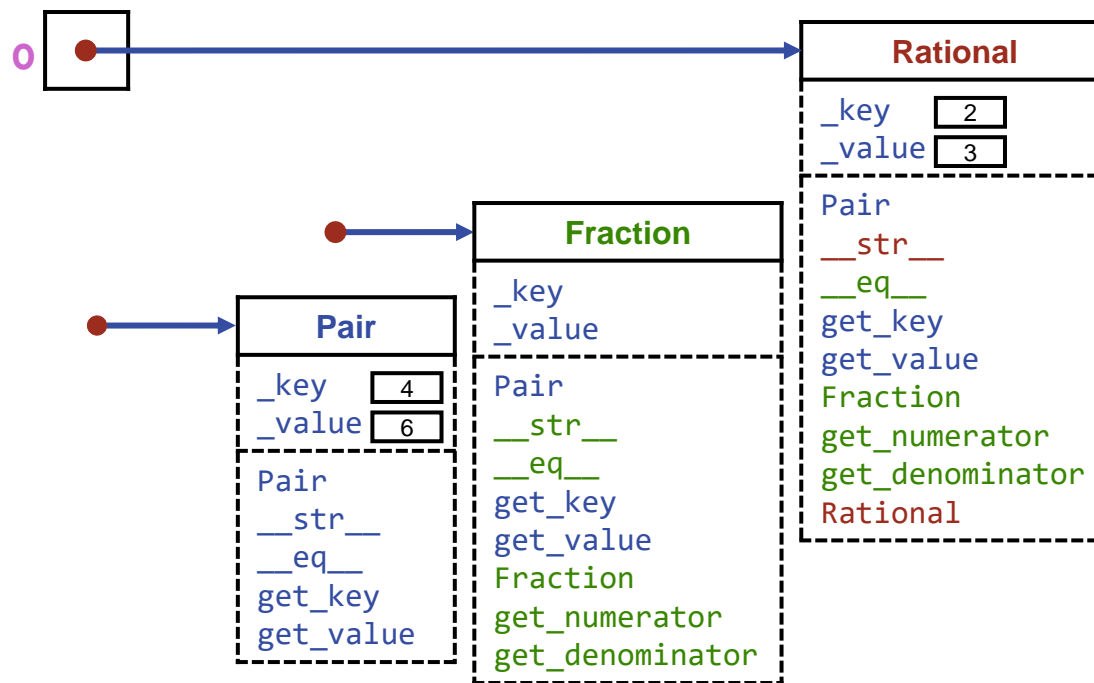
- o: object
- o = Pair(4,6)
- o = Fraction(4,6)





Subtype polymorphism:

```
o: object  
o = Pair(4,6)  
o = Fraction(4,6)  
o = Rational(4,6)
```



```

Object
 |
Pair
 |
Fraction
 |
Rational

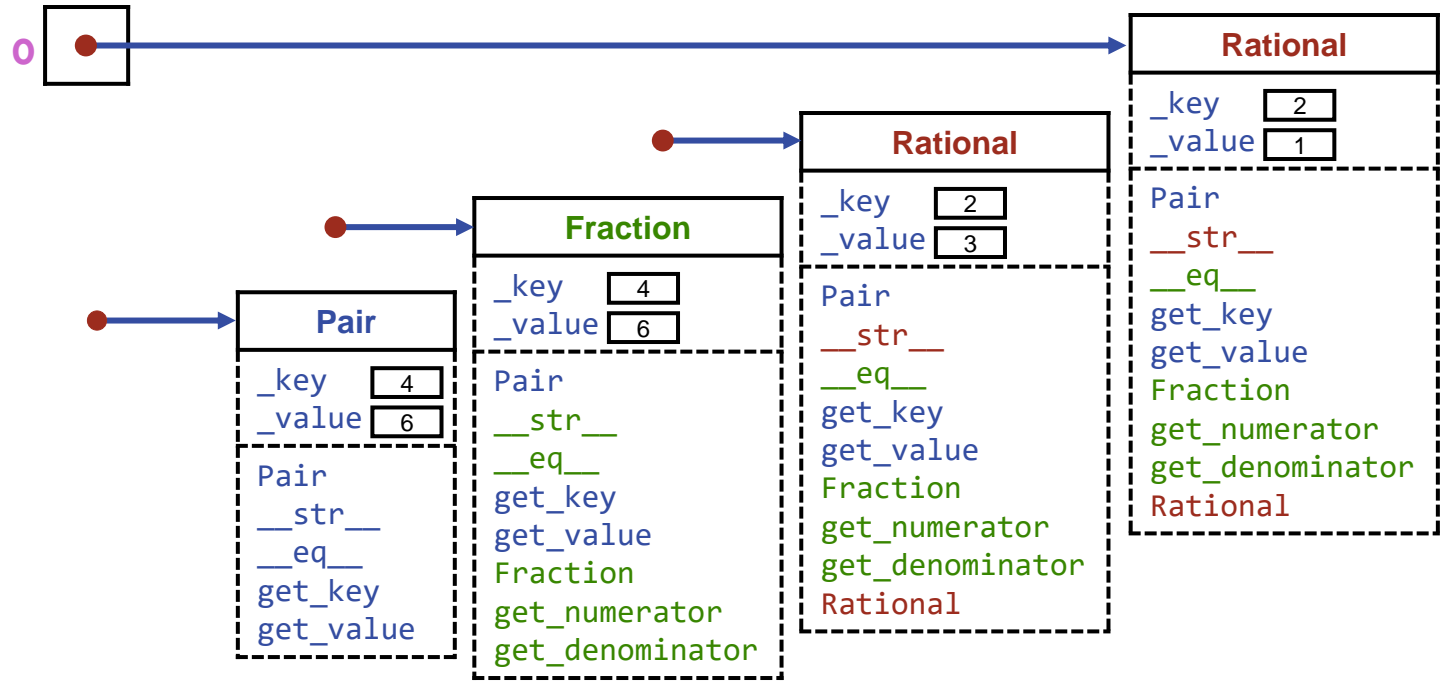
```

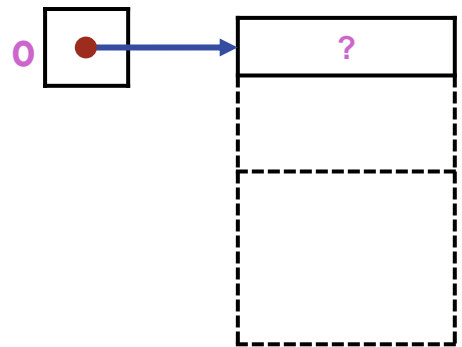
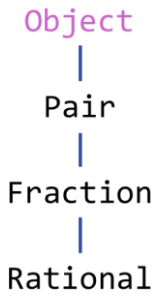
Subtype polymorphism:

```

o: object
o = Pair(4,6)
o = Fraction(4,6)
o = Rational(4,6)
o = Rational(6,3)

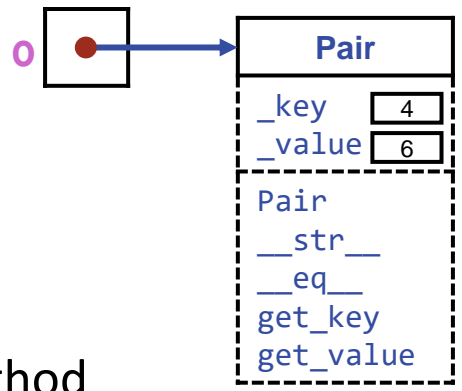
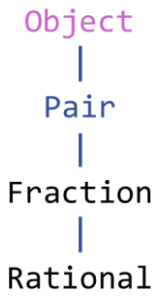
```





Dynamic method dispatch: The definition used for any given method invocation depends of the value, not the type of the variable that contains that value.

o: object

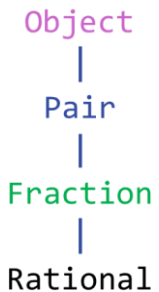


Dynamic method dispatch: The definition used for any given method invocation depends of the value, not the type of the variable that contains that value.

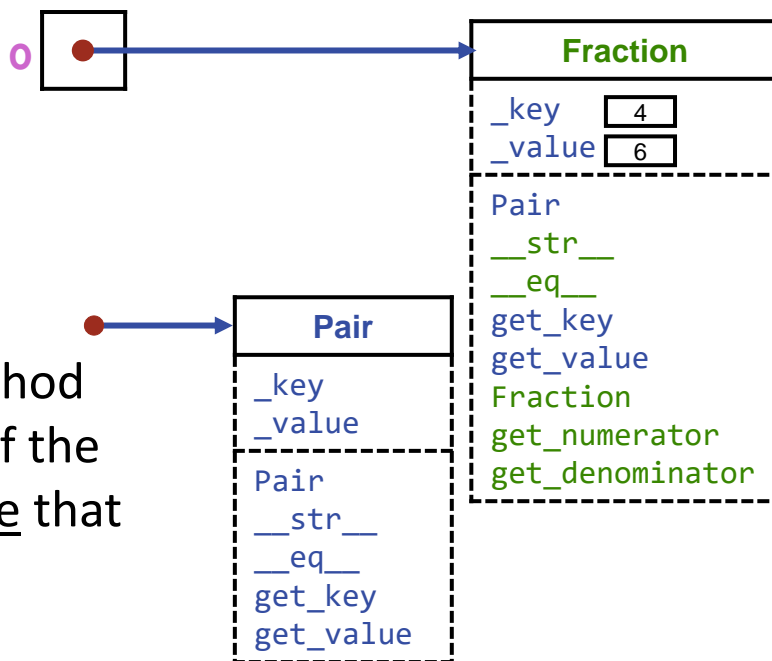
```

o: object
o = Pair(4,6);    print( o )
  
```

<4,6>



Dynamic method dispatch: The definition used for any given method invocation depends of the type of the value, not the type of the variable that contains that value.



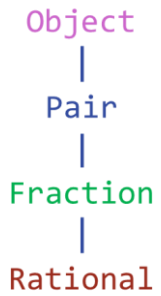
o: object

```

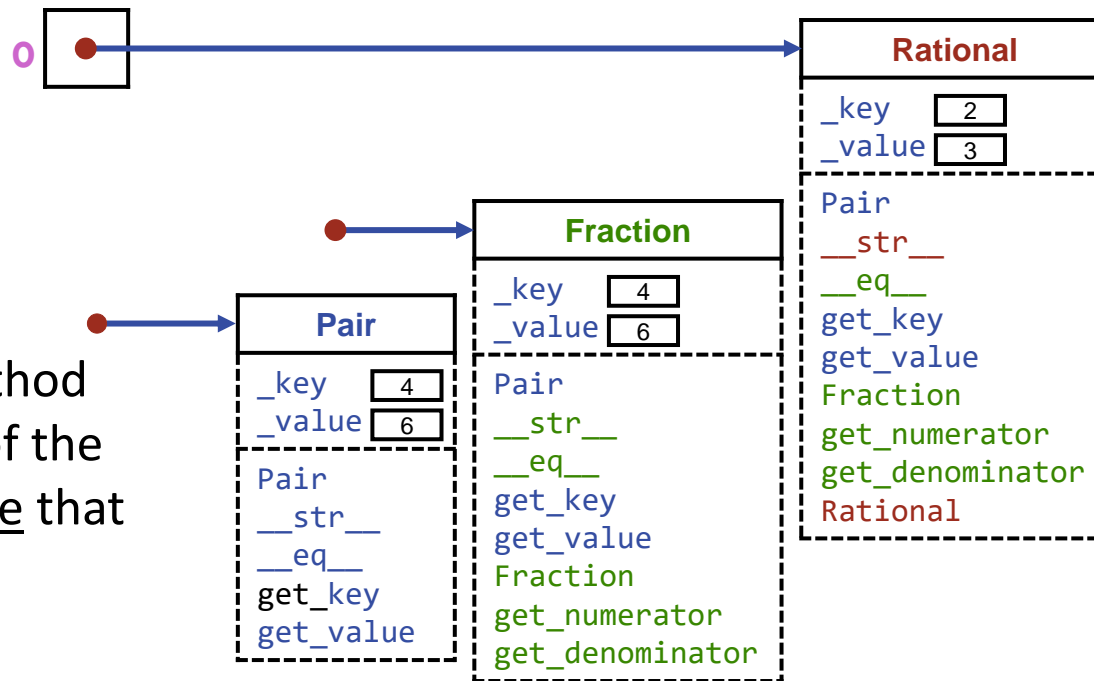
o = Pair(4,6);    print( o )
o = Fraction(4,6); print( o )
    
```

```

<4,6>
4/6
    
```

Dynamic method dispatch: The definition used for any given method invocation depends of the type of the value, not the type of the variable that contains that value.



o: object

```

o = Pair(4,6);      print( o )
o = Fraction(4,6); print( o )
o = Rational(4,6); print( o )
    
```

```

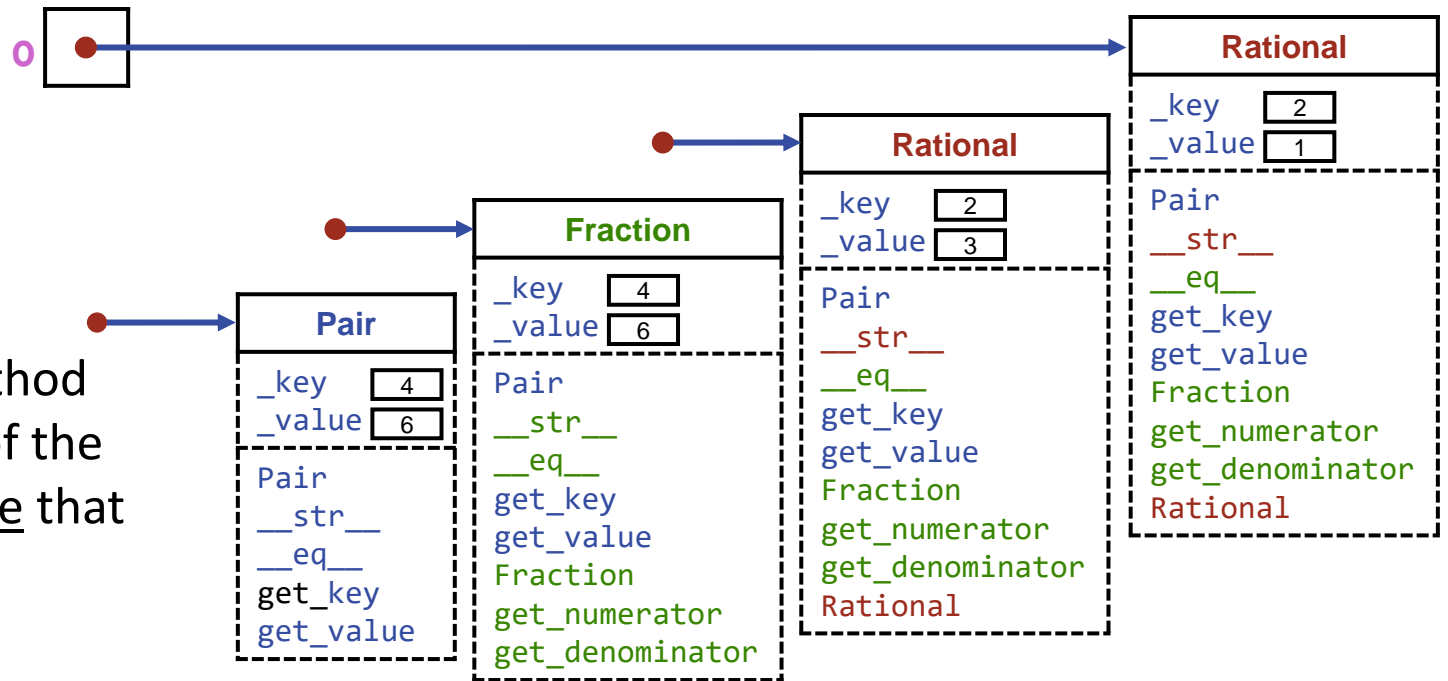
<4,6>
4/6
2/3
    
```

```

Object
 |
Pair
 |
Fraction
 |
Rational

```

Dynamic method dispatch: The definition used for any given method invocation depends of the value, not the type of the variable that contains that value.



`o: object`

```

o = Pair(4,6);      print( o )
o = Fraction(4,6); print( o )
o = Rational(4,6); print( o )
o = Rational(6,3); print( o )

```

```

<4,6>
4/6
2/3
2

```

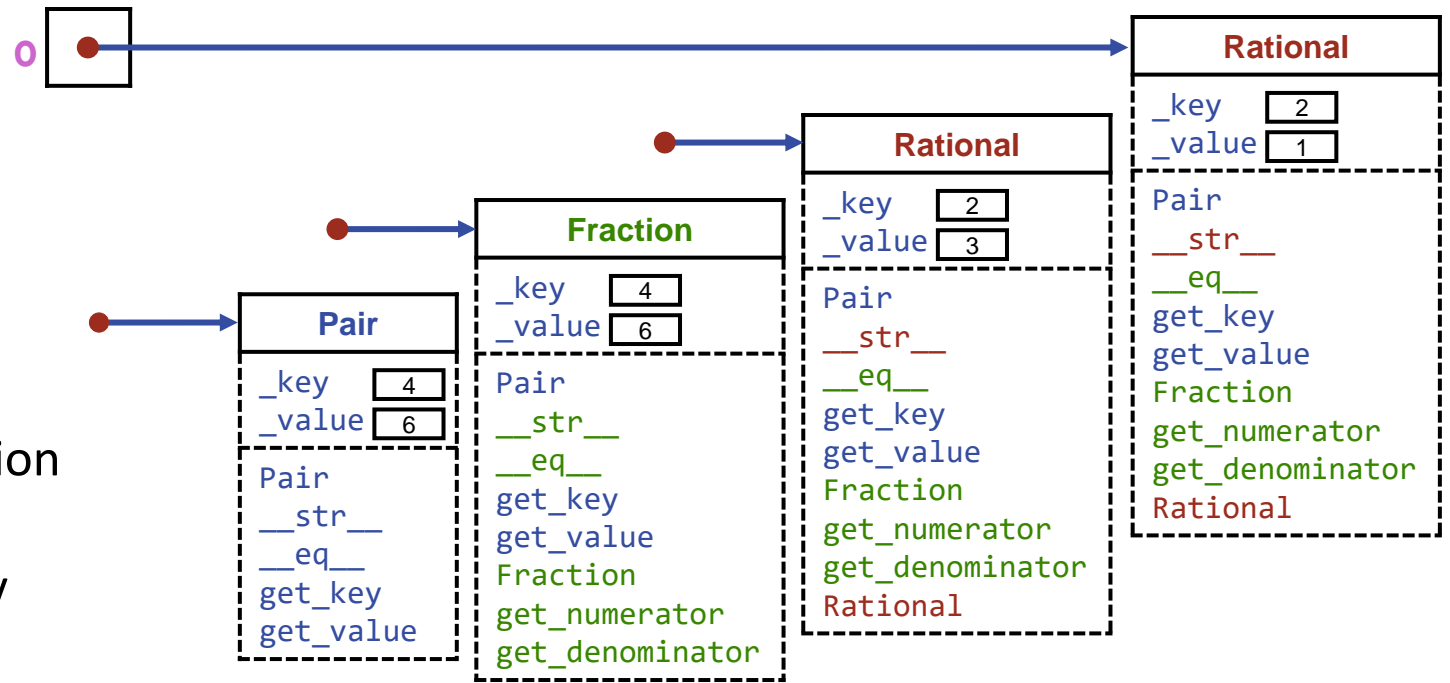
```

Object
 |
Pair
 |
Fraction
 |
Rational

```

Subtype polymorphism caveat:

If variable v has type C , a field access $v.f$, or a method invocation $v.m(\dots)$, requires that field f or method m necessarily exist in any object of type C .

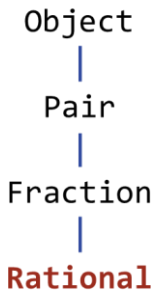


```

o: object    = Pair(4,6);    print(o.get_key())        # Statically type incorrect.
p: Pair      = Pair(4,6);    print(p.get_key())        # Statically type correct.
p            = Pair(2,3);    print(p.get_numerator())  # Statically type incorrect.
f: Fraction  = Fraction(4,6); print(f.get_numerator())  # Statically type correct.
q: Rational  = Rational(6,3); print(q.get_numerator())  # Statically type correct.

```

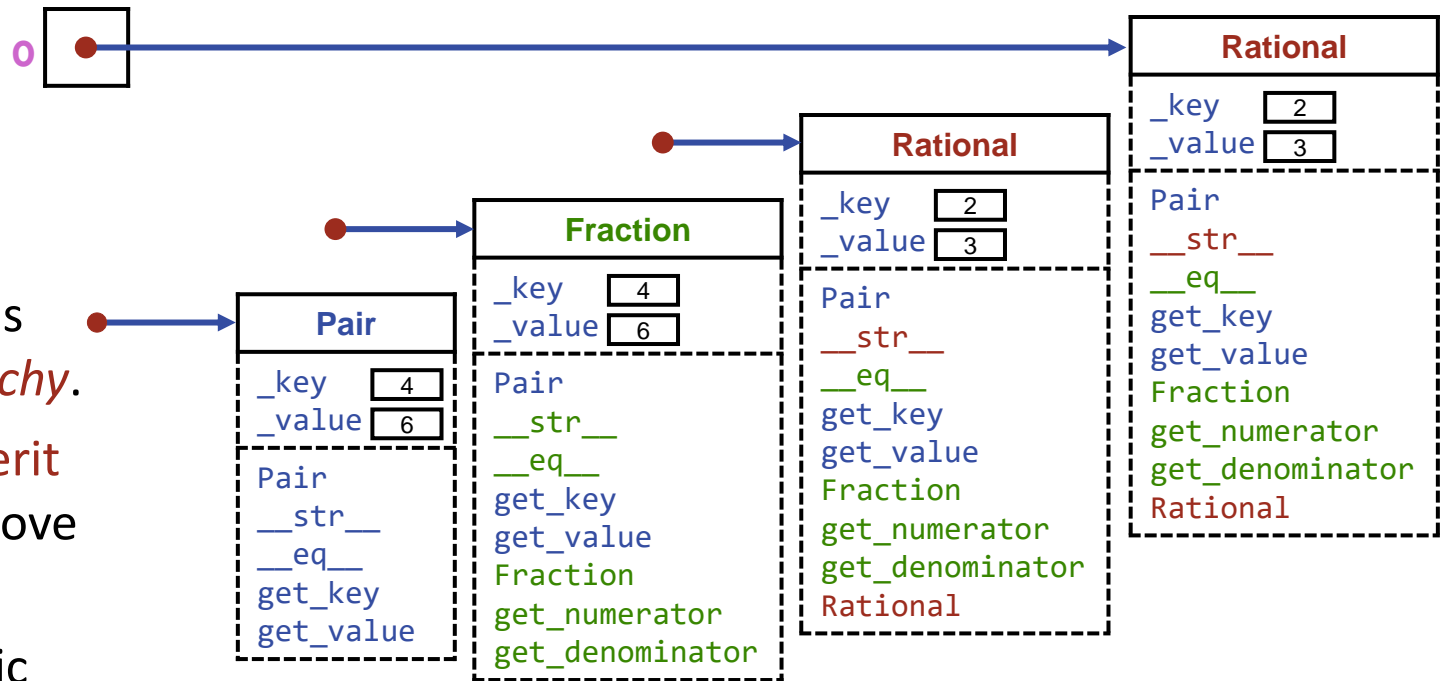
This rule holds if we wish to enforce *static type checking*, i.e., detect type errors *prior* to execution.



Inheritance: The class hierarchy is also called the *inheritance hierarchy*.

Objects of class C are said to **inherit** all fields *f* of superclasses of C above it in the hierarchy.

They also **inherit** the most specific (overriding) version of method *m* defined either in class C, or in one of C's superclasses, i.e., the first definition of *m* found in a traversal from C up to Object in the hierarchy.



Motivation: Recall that in Chapter 6 we showed how to maintain a dynamically changing collection of integers in a data structure consisting of an array `A` and an integer `size`:



We implemented each of the operations **add**, **remove**, **membership**, **multiplicity**, and **enumeration** with small code patterns. A ready facility with such patterns is important, but we remarked that writing such code directly in your program also has drawbacks:

- The collection has no single name, and thus it is not easily manipulated as one thing.
- The collection's implementation details are not hidden, and thus your program can both break the data structure's representation invariant and come to excessively depend on its details.

We address those limitations now by defining class `ArrayList`:

- References to instances of `ArrayList` can be manipulated as one thing, i.e., as objects.
- The details of an `ArrayList` are hidden using the class's visibility mechanism, which allows easy replacement of one collection implementation with another.

The implementation of Python's `list[]` type (under the hood) is essentially the same as `ArrayList`.

Technically, “hidden” is misnomer, Python only *urges* you to not rely on *protected* variables and methods.

Motivation: Recall that in Chapter 6 we showed how to maintain a dynamically changing collection of integers in a data structure consisting of an array *A* and an integer *size*:



We implemented each of the operations **add**, **remove**, **membership**, **multiplicity**, and **enumeration** with small code patterns. A ready facility with such patterns is important, but we remarked that writing such code directly in your program also has drawbacks:

- The collection has no single name, and thus it is not easily manipulated as one thing.
- The collection’s implementation details are not hidden, and thus your program can both break the data structure’s representation invariant and come to excessively depend on its details.

We address those limitations now by defining class `ArrayList`:

- References to instances of `ArrayList` can be manipulated as one thing, i.e., as objects.
- The details of an `ArrayList` are **hidden** using the class’s visibility mechanism, which allows easy replacement of one collection implementation with another.

The implementation of Python’s `list[]` type (under the hood) is essentially the same as `ArrayList`.

Guide: The implementation turns the familiar code fragments of Chapter 12 into methods, and will therefore need little additional explanation.

There is an additional benefit of turning these code fragments into the methods that was not previously mentioned:

- The data structure (and its methods) can be instantiated multiple times, i.e., you can easily have as many list objects as you want.

Each method has a docstring that provides the method's specification. Although various organizations standardize formats for such specifications, we will be less formal about their structure. Nonetheless, we aim for precision that is intended to be adequate external documentation for any client of the class and its interface.

Occasional additional notes are provided, but you should otherwise let the specifications and their implementations speak for themselves.

 **Repeatedly improve comments (and docstrings) by relentless copy editing.**

Class definition:

```
class ArrayList:
    """A list of unbounded capacity containing items of type int."""
    # Representation.
    _A: list[int]      # _A[0..size-1] is a collection of list items of type int.
    _size: int        # _size is the current number of items in the list.
                     # The current list capacity is len(_A).
```


Class definition:

```
class ArrayList:
    """A list of unbounded capacity (initially m) containing items of type int."""
    # Representation.
    _A: list[int]      # _A[0..size-1] is a collection of list items of type int.
    _size: int         # _size is the current number of items in the list.
                     # The current list capacity is len(_A).
```

The type of an ArrayList element is `int` (for now), and will be colored blue.

Integers unrelated to the type of ArrayList items will *not* be colored blue. This color coding will help you to distinguish between the type of list items, and the type of integer indices. It will also be relevant later, when we generalize the class ArrayList to be a list of items of any type.

Data representation is *protected*, i.e., hidden to clients, albeit visible to subclasses, if any.

Class definition:

```
class ArrayList:
    """A list of unbounded capacity containing items of type int."""
    # Representation.
    _A: list[int]      # _A[0..size-1] is a collection of list items of type int.
    _size: int        # _size is the current number of items in the list.
                    # The current list capacity is len(_A).
```

Class definition:

```

class ArrayList:
    """A list of unbounded capacity containing items of type int."""
    # Representation.
    _A: list[int]      # _A[0..size-1] is a collection of list items of type int.
    _size: int         # _size is the current number of items in the list.
                      # The current list capacity is len(_A).

    # Constructor.
    def __init__(self, m: int = 20) -> None:
        """
        Construct an empty list for int items, with an initial capacity  $m \geq 0$ ,
        or 20 if no  $m$  is given.
        Raise ValueError if  $m < 0$ .
        """
        if m < 0: raise ValueError("Capacity must be non-negative int")
        self._A = [0 for _ in range(m)]
        self._size = 0

```

Default value for parameter allows it to be omitted in an invocation of the constructor.

Class definition:

```
class ArrayList:
    """A list of unbounded capacity containing items of type int."""
    # Representation.
    _A: list[int] # _A[0..size-1] is a collection of list items of type int.
    _size: int # _size is the current number of items in the list.
                # The current list capacity is len(_A).

    # Constructor.
    def __init__(self, m: int = 20) -> None:
        """
        Construct an empty list for int items, with an initial capacity m>=0,
        or 20 if no m is given.
        Raise ValueError if m<0.
        """
        if m < 0: raise ValueError("Capacity must be non-negative int")
        self._A = [0 for _ in range(m)]
        self._size = 0
```

The initial values in the array are zeros, which we also color blue.

Class definition:

```
class ArrayList:
    """A list of unbounded capacity containing items of type int."""
    # Representation.
    _A: list[int]    # _A[0..size-1] is a collection of list items of type int.
    _size: int      # _size is the current number of items in the list.
                   # The current list capacity is len(_A).

    # Constructor.
    def __init__(self, m: int = 20) -> None:
        """
        Construct an empty list for int items, with an initial capacity  $m \geq 0$ ,
        or 20 if no  $m$  is given.
        Raise ValueError if  $m < 0$ .
        """
        if m < 0: raise ValueError("Capacity must be non-negative int")
        self._A = [0 for _ in range(m)]
        self._size = 0
```

A blank line after multi-line docstrings is recommended, but not done here to facilitate “slide management”.

Class definition:

```
class ArrayList:
    """A list of unbounded capacity containing items of type int."""
    # Representation.
    _A: list[int]      # _A[0..size-1] is a collection of list items of type int.
    _size: int         # _size is the current number of items in the list.
                      # The current list capacity is len(_A).

    # Constructor.
    def __init__(self, m: int = 20) -> None:
        """
        Construct an empty list for int items, with an initial capacity  $m \geq 0$ ,
        or 20 if no  $m$  is given.
        Raise ValueError if  $m < 0$ .
        """
        if m < 0: raise ValueError("Capacity must be non-negative int")
        self._A = [0 for _ in range(m)]
        self._size = 0
```

A *public* getter for the read-only field `_size`, and a *public* predicate to test for an empty list.

```
...  
  
# Size.  
def size(self) -> int:  
    """Return the number of items in the list."""  
    return self._size  
  
def is_empty(self) -> bool:  
    """Return True iff the list is empty."""  
    return self._size == 0
```

```
...

# Access.
def get(self, k: int) -> int:
    """
    Return the list item at index k.
    Raise IndexError for an out-of-bounds k.
    """
    self._check_bound_exclusive(k)
    return self._A[k]

def set(self, k: int, v: int) -> int:
    """
    Overwrite the list item at index k with v, and return the old item that was there.
    Raise IndexError for an out-of-bounds k.
    """
    self._check_bound_exclusive(k)
    old: int = self._A[k]
    self._A[k] = v
    return old
```


Raise exception if `k` is outside the bounds of the current list, **excluding** the index of the next available slot.

```
...

# Access.
def get(self, k: int) -> int:
    """
    Return the list item at index k.
    Raise IndexError for an out-of-bounds k.
    """
    self._check_bound_exclusive(k)
    return self._A[k]

def set(self, k: int, v: int) -> int:
    """
    Overwrite the list item at index k with v, and return the old item that was there.
    Raise IndexError for an out-of-bounds k.
    """
    self._check_bound_exclusive(k)
    old: int = self._A[k]
    self._A[k] = v
    return old
```

...

Insertion / Deletion.

```
def add(self, v: int, k: int = -1) -> None:
```

```
    """
```

If no k is provided, append v the end of the list, else right-shift items with indices k thru the end of the list one place, and insert v at index k.

Increase the list capacity, if necessary.

Raise IndexError on out-of-bound k.

```
    """
```

```
    if k == -1: k = self._size
```

```
    self._check_bound_inclusive(k)
```

```
    if self._size == len(self._A): self.ensure_capacity(self._size + 1)
```

```
    for j in range(self._size, k, -1): self._A[j] = self._A[j-1]
```

```
    self._A[k] = v
```

```
    self._size += 1
```

...

Insertion / Deletion.

```
def add(self, v: int, k: int = -1) -> None:
```

```
    """
```

If no k is provided, append v the end of the list, else right-shift items with indices k thru the end of the list one place, and insert v at index k.

Increase the list capacity, if necessary.

Raise IndexError on out-of-bound k.

```
    """
```

```
    if k == -1: k = self._size
```

```
    self._check_bound_inclusive(k)
```

```
    if self._size == len(self._A): self.ensure_capacity(self._size + 1)
```

```
    for j in range(self._size, k, -1): self._A[j] = self._A[j-1]
```

```
    self._A[k] = v
```

```
    self._size += 1
```

...

Insertion / Deletion.

```
def add(self, v: int, k: int = -1) -> None:
```

```
    """
```

If no k is provided, append v the end of the list, else right-shift items with indices k thru the end of the list one place, and insert v at index k.

Increase the list capacity, if necessary.

Raise IndexError on out-of-bound k.

```
    """
```

```
    if k == -1: k = self._size
```

```
    self._check_bound_inclusive(k)
```

```
    if self._size == len(self._A): self.ensure_capacity(self._size + 1)
```

```
    for j in range(self._size, k, -1): self._A[j] = self._A[j-1]
```

```
    self._A[k] = v
```

```
    self._size += 1
```

...

Insertion / Deletion.

```
def add(self, v: int, k: int = -1) -> None:
```

```
    """
```

If no k is provided, append v the end of the list, else right-shift items with indices k thru the end of the list one place, and insert v at index k.

Increase the list capacity, if necessary.

Raise IndexError on out-of-bound k.

```
    """
```

```
    if k == -1: k = self._size
```

```
    self._check_bound_inclusive(k)
```

```
    if self._size == len(self._A): self.ensure_capacity(self._size + 1)
```

```
    for j in range(self._size, k, -1): self._A[j] = self._A[j-1]
```

```
    self._A[k] = v
```

```
    self._size += 1
```

Raise `IndexError` exception if `k` is outside the bounds of the current list, but **allow** the index of the next available slot.

...

Insertion / Deletion.

```
def add(self, v: int, k: int = -1) -> None:
```

```
    """
```

```
        If no k is provided, append v the end of the list, else right-shift items with
        indices k thru the end of the list one place, and insert v at index k.
        Increase the list capacity, if necessary.
```

```
        Raise IndexError on out-of-bound k.
```

```
    """
```

```
        if k == -1: k = self._size
```

```
        self._check_bound_inclusive(k)
```

```
        if self._size == len(self._A): self.ensure_capacity(self._size + 1)
```

```
        for j in range(self._size, k, -1): self._A[j] = self._A[j-1]
```

```
        self._A[k] = v
```

```
        self._size += 1
```

...

```
def remove(self, k: int) -> int:
    """
    Return the list item with index k after left-shifting items with indices
    k+1 thru the end (if any) to remove the old k-th value from the list.
    Raise IndexError for an out-of-bounds k.
    """
    self._check_bound_exclusive(k)
    old: int = self._A[k]
    self._size -= 1
    for j in range(k, self._size): self._A[j] = self._A[j+1];
    return old

def remove_by_value(self, v: int) -> bool:
    """
    Return False if v is not in the list, else remove (one copy of) v from list
    and return True.
    """
    k = self.index_of(v)
    if k == -1: return False;
    else: self.remove(k); return True
```

...

Capacity.

```
def ensure_capacity(self, min_capacity: int) -> None:
```

```
    """
```

```
    Increase the list's capacity to the maximum of min_capacity or double its current
    capacity.
```

```
    N.B. Python's built-in type "list" has "array doubling" built in. We ignore that
    here for pedagogical purposes.
```

```
    """
```

```
    current_length: int = len(self._A)
```

```
    if min_capacity > current_length:
```

```
        B: list[int] = [0] * max(2 * current_length, min_capacity)
```

```
        for k in range(0, self._size): B[k] = self._A[k]
```

```
        self._A = B
```


...

Membership.

```
def index_of(self, v: int) -> int:
    """Return the index of an instance of v in the list, or -1 if there are none."""
    k: int = 0
    while (k < self._size) and (v != self._A[k]): k += 1
    if k == self._size: return -1
    else: return k

def contains(self, v: int) -> bool:
    """Return True iff the list contains (one or more copies of) v."""
    return self.index_of(v) != -1
```

```
...

# Bounds Checking.
def _check_bound_exclusive(self, k: int) -> None:
    """Raise IndexError if k is not the index of one of the list's items."""
    if (k < 0) or (k >= self._size): raise IndexError(">size")

def _check_bound_inclusive(self, k: int) -> None:
    """
    Raise IndexError if k is not the index of one of the list's items
    or the next available index for an item to be added.
    """
    if (k < 0) or (k > self._size): raise IndexError(">size")

# end of class ArrayList.
```

Unit Test: Cover **every public aspect** of the class's interface (*black-box* testing), and if you know the implementation internals, **every corner case** you can foresee (*white-box* testing).

```
# A useful utility function for the tests that follow.
def diag()-> None:
    print("size:", collection.size())
    print("is_empty:", collection.is_empty())
    print("contains 10:", collection.contains(10))
    print("contains 20:", collection.contains(20))
    print("index of 10:", collection.index_of(10))
    print("index of 20:", collection.index_of(20))
    print("-----")
```

 **Validate output thoroughly.**

Test code	Output
<pre>collection = ArrayList() print("new array list:") diag()</pre>	<pre>new array list: size: 0 is_empty: True contains 10: False contains 20: False index of 10: -1 index of 20: -1 -----</pre>
<pre>collection.add(10) print("add 10") diag()</pre>	<pre>add 10 size: 1 is_empty: False contains 10: True contains 20: False index of 10: 0 index of 20: -1 -----</pre>
<pre>collection.add(20) print("add 20") diag()</pre>	<pre>add 20 size: 2 is_empty: False contains 10: True contains 20: True index of 10: 0 index of 20: 1 -----</pre>
<pre>collection.remove_by_value(10) print("remove by value 10") diag()</pre>	<pre>remove by value 10 size: 1 is_empty: False contains 10: False contains 20: True index of 10: -1 index of 20: 0 -----</pre>

Test code (continued)	Output (continued)
<pre>collection.add(10,0) print("add 10 at index 0") diag()</pre>	<pre>add 10 at index 0 size: 2 is_empty: False contains 10: True contains 20: True index of 10: 0 index of 20: 1 -----</pre>
<pre>collection.add(15,1) print("add 15 at index 1") diag()</pre>	<pre>add 15 at index 1 size: 3 is_empty: False contains 10: True contains 20: True index of 10: 0 index of 20: 2 -----</pre>
<pre>v = collection.get(1) print("item at 1", v) diag()</pre>	<pre>item at 1 15 size: 3 is_empty: False contains 10: True contains 20: True index of 10: 0 index of 20: 2 -----</pre>
<pre>v = collection.set(1, 16) print("set:", v, "at 1 to 16") diag()</pre>	<pre>set: 15 at 1 to 16 size: 3 is_empty: False contains 10: True contains 20: True index of 10: 0 index of 20: 2 -----</pre>

Test code	Output
<pre>v = collection.get(1) print("item at 1 is:", v) diag()</pre>	<pre>item at 1 is: 16 size: 3 is_empty: False contains 10: True contains 20: True index of 10: 0 index of 20: 2 -----</pre>
<pre>collection.add(10) print("add 10") diag()</pre>	<pre>add 10 size: 1 is_empty: False contains 10: True contains 20: False index of 10: 0 index of 20: -1 -----</pre>

Unit Test: Seemingly mindless, but surprisingly effective. The skill involves ferreting out every way in which the code might fail.

- (1) Exercise every line of code to make sure it does not trigger a crash.
- (2) Visually inspect the output to confirm that it is correct.

Can you think of any cases we have missed?

Test code	Output
<pre>v = collection.get(1) print("item at 1 is:", v) diag()</pre>	<pre>item at 1 is: 16 size: 3 is_empty: False contains 10: True contains 20: True index of 10: 0 index of 20: 2 -----</pre>
<pre>collection.add(10) print("add 10") diag()</pre>	<pre>add 10 size: 1 is_empty: False contains 10: True contains 20: False index of 10: 0 index of 20: -1 -----</pre>

Unit Test: Seemingly mindless, but surprisingly effective. The skill involves ferreting out every way in which the code might fail.

- (1) Exercise every line of code to make sure it does not trigger a crash.
- (2) Visually inspect the output to confirm that it is correct.

Can you think of any cases we have missed?

Visual inspection of output is tedious, and not something you want to redo manually after every code change. It is common to automate such retests by capturing the desired output in a file to which new output can be compared automatically after each change.

Enumeration of rationals: Recall this incomplete code example from Chapter 6.

```
# Output reduced positive fractions, i.e., positive rationals.
# set reduced = { }
d = 0
while True:
    r = d;
    for c in range(0, d+1):
        # Let z be the reduced form of the fraction (r+1)/(c+1).
        g: int = gcd(r + 1, c + 1);
        # z: rational = <(r+1)/g, (c+1)/g>

        if # z-is-not-an-element-of-reduced :
            # print(z)
            # reduced = reduced U {z}
    r -= 1
d += 1
```

Enumeration of rationals: We can adopt `Rational` as the type of the rational z .

```
# Output reduced positive fractions, i.e., positive rationals.
# set reduced = { }
d = 0
while True:
    r = d;
    for c in range(0, d+1):
        # Let z be the reduced form of the fraction (r+1)/(c+1).
        z: Rational = Rational(r+1, c+1)

        if # z-is-not-an-element-of-reduced :
            # print(z)
            # reduced = reduced U {z}
    r -= 1
d += 1
```


We would like to adopt `ArrayList` as the type of the set `reduced`, but cannot do so because, as currently written, it is a collection of `int` items, not `Rational` items.

Enumeration of rationals: We can adopt `Rational` as the type of rational `z`.

```
# Output reduced positive fractions, i.e., positive rationals.
# set reduced = { }
d = 0
while True:
    r = d;
    for c in range(0, d+1):
        # Let z be the reduced form of the fraction (r+1)/(c+1).
        z: Rational = Rational(r+1, c+1)

        if # z-is-not-an-element-of-reduced :
            # print(z)
            # reduced = reduced U {z}
        r -= 1
    d += 1
```

Enumeration of rationals: We need an `ArrayList` of `Rational` items.

This could be done by:

- Cloning the `ArrayList` of `int` implementation, and adapting the clone to be a collection of `Rational` elements (ugh!), or
- Parameterizing `ArrayList` to be `ArrayList[E]`, a collection of elements of arbitrary object type `E`, and then instantiating it as `ArrayList[Rational]`, a collection of `Rational` elements (far better!).

A class definition that is parametrized by a type is called a *generic class*.

An array of null pointers cast to type **E** is created.

The type of an ArrayList item is parameterized as **E**.

Generic class definition:

```
class ArrayList[E]:
    """A list of unbounded capacity containing items of type E."""
    _A: list[E]      # _A[0..size-1] is a collection of items of type E.
    _size: int      # _size is the current number of items in the list.
                   # The current list capacity is len(_A).

# Constructor.
def __init__(self, m: int = 20) -> None:
    """
    Construct an empty list with capacity m>=0, or 20 if no m is provided.
    Raise ValueError for m<0.
    """
    if m < 0: raise ValueError("Capacity must be non-negative int")
    self._A = [cast(E, None) for _ in range(m)]
    self._size = 0
```

We will not repeat the definitions of every method, but will let these two illustrate what is needed. Essentially, every (blue) **int** is turned into a type parameter **E**.

```
...  
  
# Access.  
def get(self, k: int) -> E:  
    """  
    Return the list item at index k.  
    Raise IndexError for an out-of-bounds k.  
    """  
    self._check_bound_exclusive(k)  
    return self._A[k]  
  
def set(self, k: int, v: E) -> E:  
    """  
    Overwrite the list item at index k with v, and return the old item that was there.  
    Raise IndexError for an out-of-bounds k.  
    """  
    self._check_bound_exclusive(k)  
    old: E = self._A[k]  
    self._A[k] = v  
    return old  
  
...
```

A non-obvious subtlety in method `remove` involves an erasure step that assists in the efficient management of storage. This is explained in the [Garbage Collection discussion](#), later.

```
...  
  
# Insertion / Deletion.  
...  
  
def remove(self, k: int) -> E:  
    """  
    Return the list item with index k after left-shifting items with indices  
    k+1 thru the end (if any) to remove the old k-th value from the list.  
    Raise IndexError for an out-of-bounds k.  
    """  
    self._check_bound_exclusive(k)  
    old: E = self._A[k]  
    self._size -= 1  
    for j in range(k, self._size): self._A[j] = self._A[j+1];  
    self._A[self._size] = cast(E, None)    # Garbage-collection assist.  
    return old  
  
...
```

Enumeration of rationals: Returning to the incomplete code for enumerating rationals.

```
# Output reduced positive fractions, i.e., positive rationals.
# set reduced = { }
d = 0
while True:
    r = d
    for c in range(0, d+1):
        # Let z be the reduced form of the fraction (r+1)/(c+1)
        z: Rational = Rational(r + 1, c + 1)

        if # z-is-not-an-element-of-reduced :
            # print(z)
            # reduced = reduced U {z}
    r -= 1
d += 1
```

Enumeration of rationals: We declare `reduced` to have type `ArrayList[Rational]`.

```
# Output reduced positive fractions, i.e., positive rationals.
reduced = ArrayList[Rational]()
d = 0
while True:
    r = d
    for c in range(0, d+1):
        # Let z be the reduced form of the fraction (r+1)/(c+1)
        z: Rational = Rational(r + 1, c + 1)

        if not reduced.contains(z):
            print(z)
            reduced.add(z)
    r -= 1
d += 1
```

Some pedants would say that the variable `reduced` should be adorned with the type annotation “: `ArrayList[Rational]`”, but this seems a little, well, pedantic.

Enumeration of rationals: We declare `reduced` to have type `ArrayList[Rational]`.

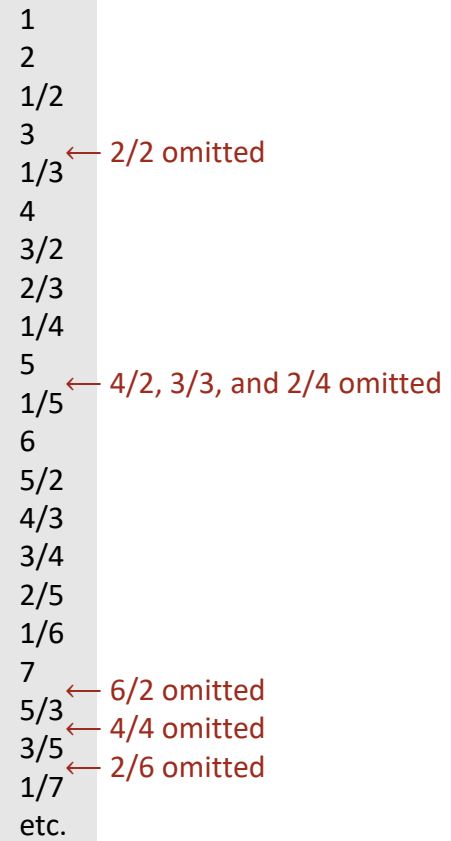
```
# Output reduced positive fractions, i.e., positive rationals.
reduced = ArrayList[Rational]()
d = 0
while True:
    r = d
    for c in range(0, d+1):
        # Let z be the reduced form of the fraction (r+1)/(c+1)
        z: Rational = Rational(r + 1, c + 1)

        if not reduced.contains(z):
            print(z)
            reduced.add(z)
    r -= 1
    d += 1
```


Enumeration of rationals: and obtain the correct output.

```
1
2
1/2
3 ← 2/2 omitted
1/3
4
3/2
2/3
1/4
5 ← 4/2, 3/3, and 2/4 omitted
1/5
6
5/2
4/3
3/4
2/5
1/6
7 ← 6/2 omitted
5/3 ← 4/4 omitted
3/5 ← 2/6 omitted
1/7
etc.
```

Enumeration of rationals: and obtain the correct output.



1
2
1/2
3 ← 2/2 omitted
1/3
4
3/2
2/3
1/4
5
1/5 ← 4/2, 3/3, and 2/4 omitted
6
5/2
4/3
3/4
2/5
1/6
7
5/3 ← 6/2 omitted
3/5 ← 4/4 omitted
1/7 ← 2/6 omitted
etc.



Object
|
Pair
|
Fraction
|
Rational

Class definition: Recall the definition of class `Pair`.

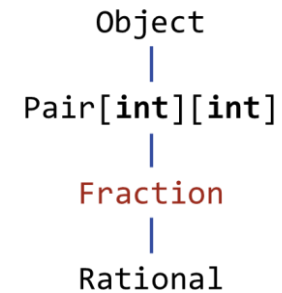
```
class Pair:
    # Representation
    _key: int
    _value: int

    # Constructor.
    def __init__(self, k: int, v: int) -> None:
        self._key = k; self._value = v

    # Access.
    def get_key(self) -> int: return self._key
    def get_value(self) -> int: return self._value
```

Generic class definition: It, too, can be made **generic** so we can have pairs of any types.

```
class Pair [K, V]:  
  # Representation  
  _key: K  
  _value: V  
  
  # Constructor.  
  def __init__(self, k: K, v: V) -> None:  
    self._key = k; self._value = v  
  
  # Access.  
  def get_key(self) -> K: return self._key  
  def get_value(self) -> V: return self._value
```



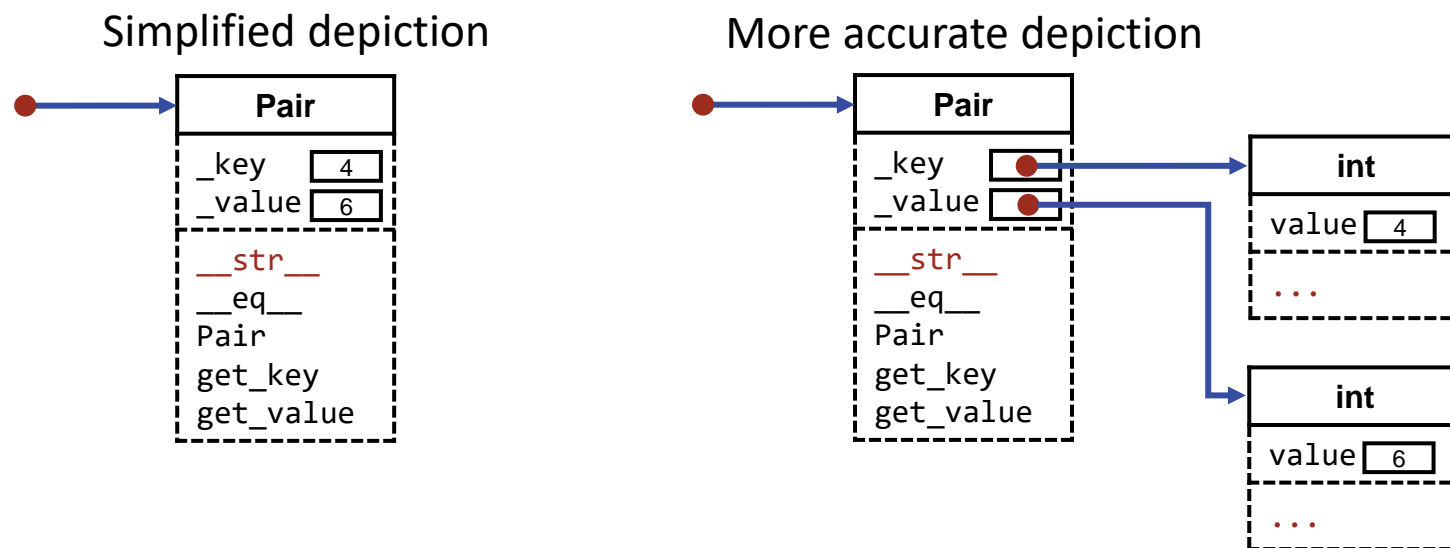
Generic class instantiation: Obtain the original `Fraction` by *generic class instantiation*.

```
class Fraction(Pair[int][int]):
    # Constructor.
    def __init__(self, numerator: int, denominator: int) -> Fraction:
        super(numerator, denominator); # Apply the Pair constructor.
        assert denominator != 0, "0 denominator"

    # Access.
    def get_numerator(self) -> int: return self._key
    def get_denominator(self) -> int: return self._value
```

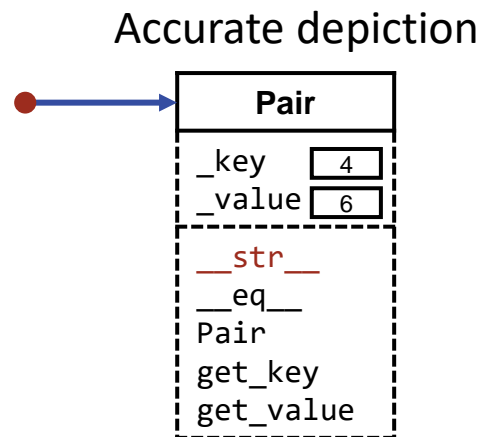
Uniformity:

- In some languages, e.g., **Python**, all values are uniformly objects of some class, and each value is accessed via a reference.
- The object reference (●) has a standard size, but the object itself doesn't.
- In such languages, even values of basic types like **int** and **bool** are objects.



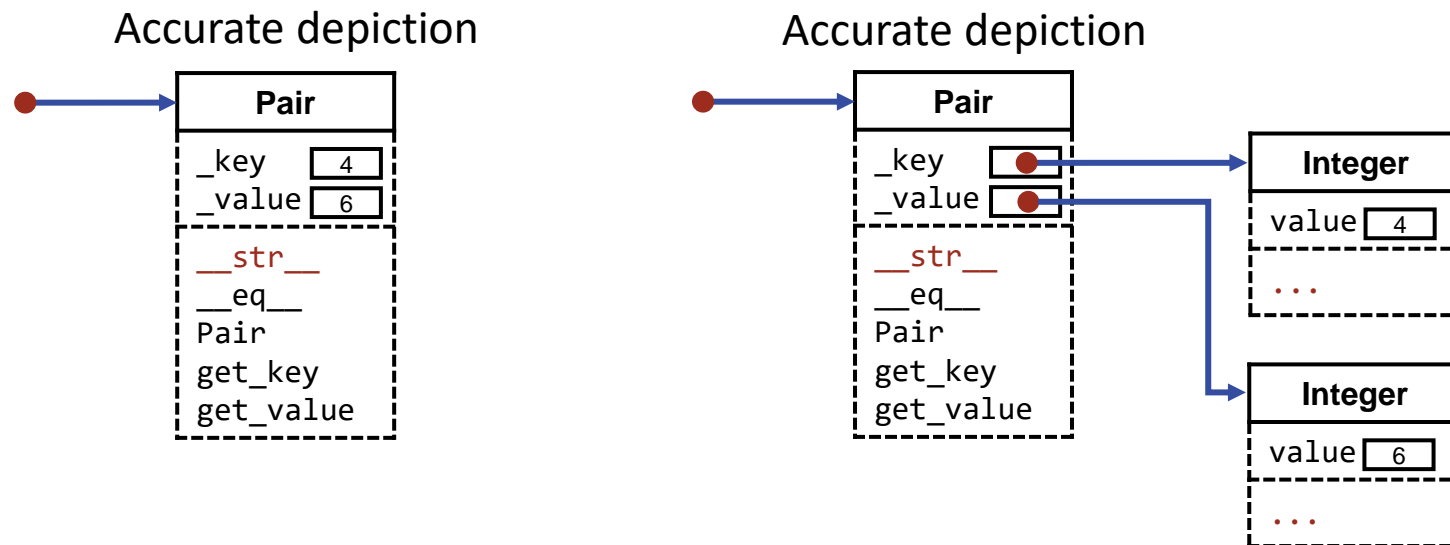
Uniformity:

- Other languages, e.g., **Java**, distinguish between *primitive values* and objects of a class.
- Primitive values, e.g., values of types **int** and **boolean**, fit conveniently into variables of standard sizes, and are not accessed via a reference.
- In such languages, the depiction characterized as “simplified” is actually accurate.



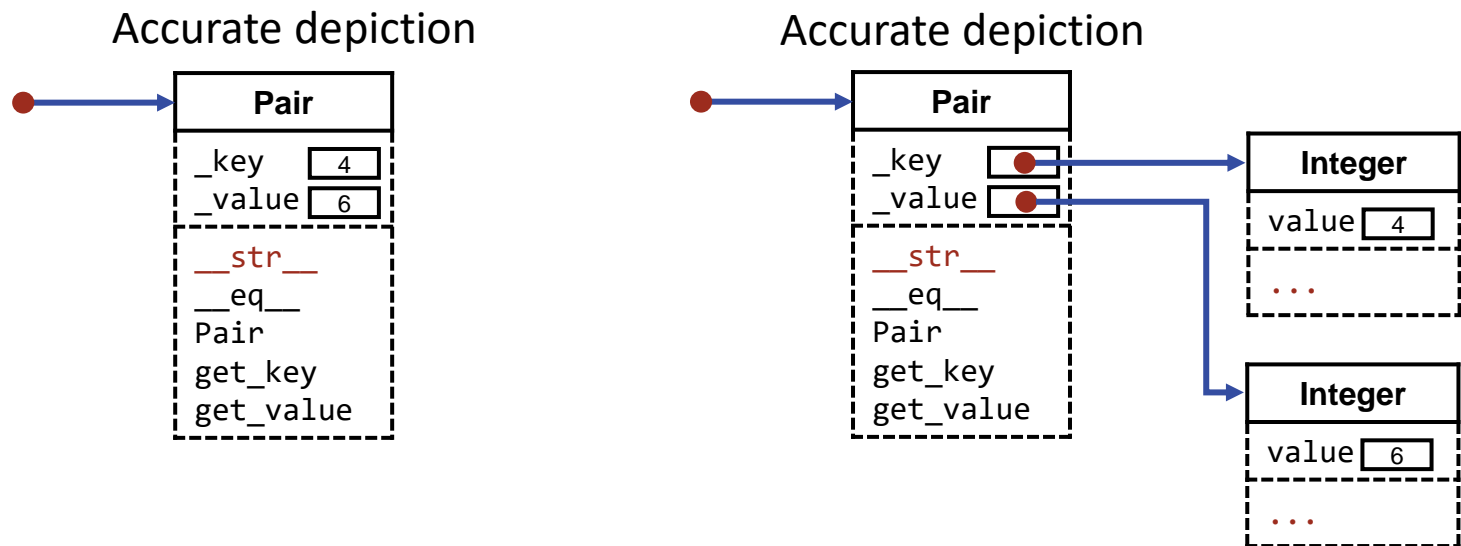
Uniformity:

- In the interest of efficiency, but at the expense of complexity, **Java** offers two worlds, one in which values of types like `int` and `boolean` are *primitive*, and the other in which there are *object versions* of such values (of types `Integer` and `Boolean`) known as *boxed* integers and `Booleans`.
- Crossing back and forth between the two worlds is a bit complicated, but is ameliorated by features known as *auto-boxing* and *auto-unboxing* (not further described).



Uniformity:

- An advantage of a language in which **all** values are objects is that generic classes can be instantiated with **any** types. In contrast, in a language that distinguished between primitive values and objects, generic classes can not be instantiated with primitive types such as `int` and `boolean`.



Polymorphism: Four kinds have been mentioned.

- **Subtype polymorphism**, where an object of one class is treated as an instance of any of its superclasses. Thus, a variable declared to have a given class as its type may contain a value of that class, or of any of its subclasses. Dynamic dispatch selects the appropriate code for a method invocation based on the specific type of the given value.
- **Parametric polymorphism**, where a class definition is abstracted with respect to one or more class parameters, resulting in a generic class, which can be viewed as a cookie cutter that stamps out classes (i.e., generic-class instances).
- **Conversion**, where an expression of one type occurs in a context that expects a value of a different type, and it is converted (either implicitly or explicitly) to the required type. Type casting is an example of such a conversion. Another term for conversion is coercion.
- **Overloading**, where different methods have the same name, and the appropriate definition is chosen based on the number and types of arguments in the invocation. Java has method overloading, but Python does not.

e.g., the object constructed by `Rational(2,3)` can be treated as a `Rational`, `Fraction`, `Pair`, or `Object`.

Polymorphism: Four kinds have been mentioned.

- **Subtype polymorphism**, where an object of one class is treated as an instance of any of its superclasses. Thus, a variable declared to have a given class as its type may contain a value of that class, or of any of its subclasses. Dynamic dispatch selects the appropriate code for a method invocation based on the specific type of the given value.
- **Parametric polymorphism**, where a class definition is abstracted with respect to one or more class parameters, resulting in a generic class, which can be viewed as a cookie cutter that stamps out classes (i.e., generic-class instances).
- **Conversion**, where an expression of one type occurs in a context that expects a value of a different type, and it is converted (either implicitly or explicitly) to the required type. Type casting is an example of such a conversion. Another term for conversion is coercion.
- **Overloading**, where different methods have the same name, and the appropriate definition is chosen based on the number and types of arguments in the invocation. Java has method overloading, but Python does not.

e.g., a variable declared to have type `Fraction` can be assigned a `Fraction` or `Rational`, but it cannot be assigned a `Pair` or `Object`.

Polymorphism: Four kinds have been mentioned.

- **Subtype polymorphism**, where an object of one class is treated as an instance of any of its superclasses. Thus, a variable declared to have a given class as its type may contain a value of that class, or of any of its subclasses. Dynamic dispatch selects the appropriate code for a method invocation based on the specific type of the given value.
- **Parametric polymorphism**, where a class definition is abstracted with respect to one or more class parameters, resulting in a generic class, which can be viewed as a cookie cutter that stamps out classes (i.e., generic-class instances).
- **Conversion**, where an expression of one type occurs in a context that expects a value of a different type, and it is converted (either implicitly or explicitly) to the required type. Type casting is an example of such a conversion. Another term for conversion is coercion.
- **Overloading**, where different methods have the same name, and the appropriate definition is chosen based on the number and types of arguments in the invocation. Java has method overloading, but Python does not.

e.g., the code executed for `__str__` depends on the type of the object, e.g., `Rational`.

Polymorphism: Four kinds have been mentioned.

- **Subtype polymorphism**, where an object of one class is treated as an instance of any of its superclasses. Thus, a variable declared to have a given class as its type may contain a value of that class, or of any of its subclasses. Dynamic dispatch selects the appropriate code for a method invocation based on the specific type of the given value.
- **Parametric polymorphism**, where a class definition is abstracted with respect to one or more class parameters, resulting in a generic class, which can be viewed as a cookie cutter that stamps out classes (i.e., generic-class instances).
- **Conversion**, where an expression of one type occurs in a context that expects a value of a different type, and it is converted (either implicitly or explicitly) to the required type. Type casting is an example of such a conversion. Another term for conversion is coercion.
- **Overloading**, where different methods have the same name, and the appropriate definition is chosen based on the number and types of arguments in the invocation. Java has method overloading, but Python does not.

e.g., `ArrayList[E]` or `Pair[K, V]`.

Polymorphism: Four kinds have been mentioned.

- **Subtype polymorphism**, where an object of one class is treated as an instance of any of its superclasses. Thus, a variable declared to have a given class as its type may contain a value of that class, or of any of its subclasses. Dynamic dispatch selects the appropriate code for a method invocation based on the specific type of the given value.
- **Parametric polymorphism**, where a class definition is abstracted with respect to one or more class parameters, resulting in a generic class, which can be viewed as a cookie cutter that stamps out classes (i.e., generic-class instances).
- **Conversion**, where an expression of one type occurs in a context that expects a value of a different type, and it is converted (either implicitly or explicitly) to the required type. Type casting is an example of such a conversion. Another term for conversion is coercion.
- **Overloading**, where different methods have the same name, and the appropriate definition is chosen based on the number and types of arguments in the invocation. Java has method overloading, but Python does not.

e.g., `ArrayList[Rational]` or `Pair[int,int]`.

Polymorphism: Four kinds have been mentioned.

- **Subtype polymorphism**, where an object of one class is treated as an instance of any of its superclasses. Thus, a variable declared to have a given class as its type may contain a value of that class, or of any of its subclasses. Dynamic dispatch selects the appropriate code for a method invocation based on the specific type of the given value.
- **Parametric polymorphism**, where a class definition is abstracted with respect to one or more class parameters, resulting in a generic class, which can be viewed as a cookie cutter that stamps out classes (i.e., generic-class instances).
- **Conversion**, where an expression of one type occurs in a context that expects a value of a different type, and it is converted (either implicitly or explicitly) to the required type. Type casting is an example of such a conversion. Another term for conversion is coercion.
- **Overloading**, where different methods have the same name, and the appropriate definition is chosen based on the number and types of arguments in the invocation. Java has method overloading, but Python does not.

e.g., `cast(E, None)` in the constructor `__init__` of the generic class `ArrayList[E]`.

Polymorphism: Four kinds have been mentioned.

- **Subtype polymorphism**, where an object of one class is treated as an instance of any of its superclasses. Thus, a variable declared to have a given class as its type may contain a value of that class, or of any of its subclasses. Dynamic dispatch selects the appropriate code for a method invocation based on the specific type of the given value.
- **Parametric polymorphism**, where a class definition is abstracted with respect to one or more class parameters, resulting in a generic class, which can be viewed as a cookie cutter that stamps out classes (i.e., generic-class instances).
- **Conversion**, where an expression of one type occurs in a context that expects a value of a different type, and it is converted (either implicitly or explicitly) to the required type. Type casting is an example of such a conversion. Another term for conversion is coercion.
- **Overloading**, where different methods have the same name, and the appropriate definition is chosen based on the number and types of arguments in the invocation. Java has method overloading, but Python does not.

Technically, `ArrayList[E]` has only one constructor, but when the initial capacity parameter is omitted, a default capacity is used. This simulates method overloading. Similarly, there is only one `add` method, but when the index is omitted, the item is added to the end, which simulates a second version of `add`.

Polymorphism: Four kinds have been mentioned.

- **Subtype polymorphism**, where an object of one class is treated as an instance of any of its superclasses. Thus, a variable declared to have a given class as its type may contain a value of that class, or of any of its subclasses. Dynamic dispatch selects the appropriate code for a method invocation based on the specific type of the given value.
- **Parametric polymorphism**, where a class definition is abstracted with respect to one or more class parameters, resulting in a generic class, which can be viewed as a cookie cutter that stamps out classes (i.e., generic-class instances).
- **Conversion**, where an expression of one type occurs in a context that expects a value of a different type, and it is converted (either implicitly or explicitly) to the required type. Type casting is an example of such a conversion. Another term for conversion is coercion.
- **Overloading**, where different methods have the same name, and the appropriate definition is chosen based on the number and types of arguments in the invocation. Java has method overloading, but Python does not.

Visibility: Three kinds have been mentioned.

- **Public**, where names (with no leading underscore) are globally visible.
- **Private**, where names (with leading double underscores) are not visible outside of the class or its subclasses.
- **Protected**, where names (with a leading single underscore) are visible within subclasses, but should not be otherwise accessed outside the class even though this rule is not enforced.

We have not illustrated *private* names because their double-underscore prefixes render such code less readable. Nonetheless, for *strict enforcement of information hiding*, *private* names should, in fact, use double underscore prefixes.

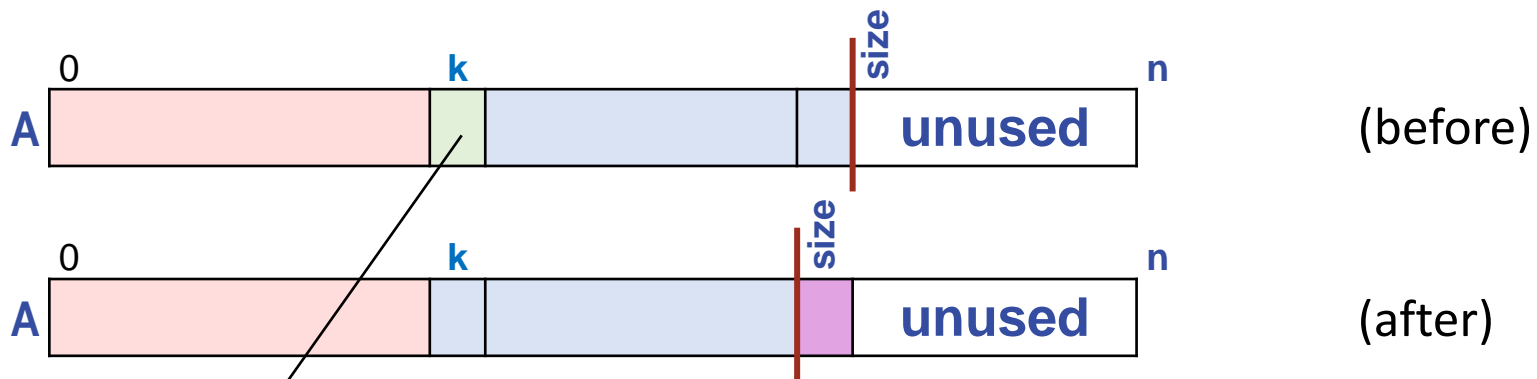
Static Type Checking:

Vital as different types proliferate. Recommendation:

- Use type annotations generously.
- Use **mypy** to check type correctness before execution.

Garbage Collection. An object dies when it can no longer be accessed in the program.

- Objects consume space in computer memory.
- Space consumed by objects that can no longer be accessed can be reclaimed automatically by a mechanism (that runs behind the scene) called *garbage collection*.
- Normally, you don't have to think about such matters. However, you should be aware that retaining a gratuitous reference to an object can cause it to be needlessly retained.
- By itself, one such object is no big concern. But if it is at the beginning of a chain of references from one object to another, then that one gratuitous reference can be the cause of an unbounded number of needlessly-retained other objects, which is of concern.
- This is why we make sure that an `ArrayList[E]` retains no gratuitous references to objects in the unused suffix of the array.
- We explain how this works next. It is a bit subtle, but is instructive.

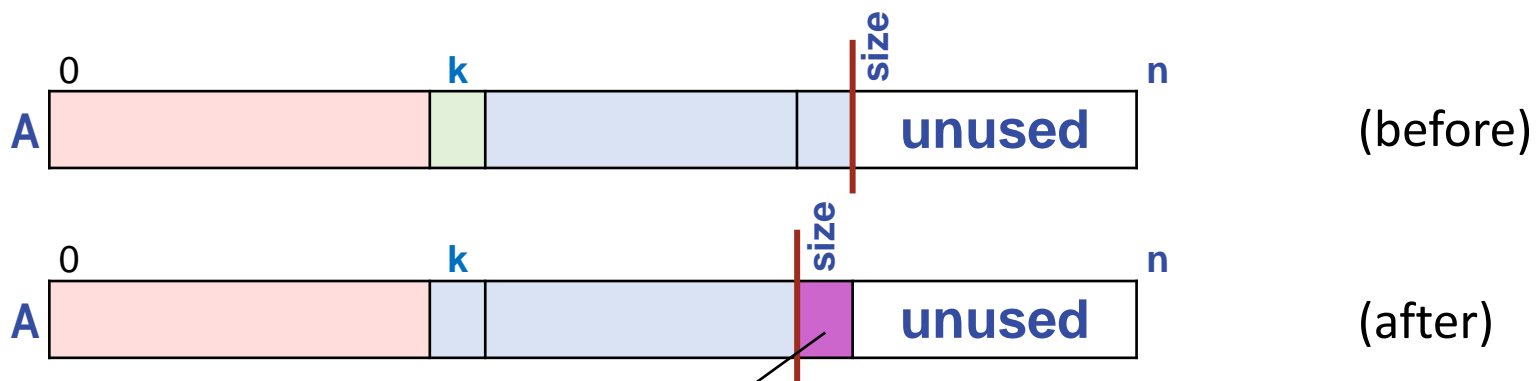


Garbage Collection. Recall the definition of `remove` in `ArrayList[E]`.

```
def remove(self, k: int) -> E:
    """ ... """

    self._check_bound_exclusive(k)
    old: E = self._A[k]
    self._size -= 1
    for j in range(k, self._size): self._A[j] = self._A[j+1];
    self._A[size] = cast(E, None) # Garbage-collection assist.
    return old
```

The left shift of (blue) values overwrites the (green) k-th value that is being removed from the collection. It was a reference to some object of arbitrary size and complexity, and if this had been the only reference to that object, it could now be garbage collected. But the value being removed at `A[k]` is not the issue, as that reference is being overwritten.



Garbage Collection. Recall the definition of `remove` in `ArrayList[E]`.

```
def remove(self, k: int) -> E:
    """ ... """

    self._check_bound_exclusive(k)
    old: E = self._A[k]
    self._size -= 1
    for j in range(k, self._size): self._A[j] = self._A[j+1];
    self._A[size] = cast(E, None)    # Garbage-collection assist.
    return old
```

The issue is the *last* (blue) value in the collection, which was originally in $A[size-1]$, and that has now been left-shifted one place. A copy of that value remains in $A[size]$, the first element of the unused array suffix. It is that violet copy that we must nullify. Note that the object referred to by the violet reference can not yet be collected because a reference to it remains in $A[size-1]$. However, if and when *that* reference is removed or is overwritten, the object in question will *then* be collectable by virtue of our having nullified the problematic copy in $A[size]$.

Alternative implementation of (some of) the interface of ArrayList[E]

A class that hides all of its implementation details, and only exposes its public methods is known as an *abstract data type*. The names, return types, and parameter types are known as the class's *interface*.

Writing code with abstract data types permits the (relatively easy) replacement of one implementation with an another, a decided advantage.

We illustrate this by defining HashSet[E], an alternative to ArrayList[E]. Because we use Python's built-in set data type to do so, the resulting implementation is rather trivial. But more importantly, we can embrace the alternative by making just a one-line change in the application! A timing study shows the substantial benefit of hash tables over lists.

(We note that because the set data type does not support indexed collections of elements, only part of the ArrayList[E] interface is implemented. More precisely, the entire interface is implemented, but the unavailable operations, if invoked, result in run-time exceptions.)

Hash Set implementation of (some of) the same interface as ArrayList[E]

```
from typing import NoReturn
class HashSet[E]:
    """Generic HashSet containing items of type E."""
    # Representation.
    _A: set[E]      # _A is the set of items.

    # Constructor.
    def __init__(self, m: int = 20) -> None:
        """A set of unbounded capacity containing items of type E."""
        if m < 0: raise ValueError("Capacity must be non-negative int")
        self._A = set[E]()

    # Size.
    def size(self) -> int:
        """Return the number of items in the set."""
        return len(self._A)

    def is_empty(self) -> bool:
        """Return True iff the set is empty."""
        return len(self._A) == 0
    ...
```



```
...

# Access.
def get(self, k: int) -> NoReturn:
    """Unsupported."""
    assert False, "get not supported"

def set(self, k: int, v: E) -> NoReturn:
    """Unsupported."""
    assert False, "set not supported"

# Insertion / Deletion.
def add(self, v: E, k: int = -1) -> None:
    """
    If no k is provided, insert v into the set, else raise an assertion
    exception "add at index not supported".
    """
    if k != -1:
        raise assertion exception, "add at index not supported"
    self._A.add(v)
```

```
...  
  
def remove(self, k: int) -> NoReturn:  
    """Unsupported."""  
    assert False, "remove (by index) not supported"  
  
def remove_by_value(self, v: E) -> bool:  
    """Return False if v is not in set, else remove v from set and return True."""  
    try:  
        self._A.remove(v)  
        return True  
    except ValueError:  
        return False  
  
# Capacity.  
def ensure_capacity(self, min_capacity int) -> None:  
    """A superfluous operation purporting to increase the set's capacity."""  
    pass
```

```
...  
  
# Membership.  
def index_of(self, v: E) -> NoReturn:  
    """Unsupported."""  
    assert False, "index_of not supported"  
  
def contains(self, v: E) -> bool:  
    """Return True iff the set contains v."""  
    return v in self._A
```

Hash Functions: The implementation of `HashSet[E]` requires that its `E`'s have a hash function, but `Rational` (as we have defined it) does not have one. The following simple hash function just sums the hashes provided by the `int` numerator and denominator fields:

```
class Rational:
    ...
    # Hash Function
    def __hash__(self):
        return hash(self._key) + hash(self._value)
    ...
```

It would be more general-purpose if `Rational` were to inherit a hash function from `Pair[K, V]`, which we could define when `K` and `V` themselves have hash functions. Unfortunately, this combines with complexities of generic-class type parameters, and gets us over our heads in fine points, which we will finesse.

Enumeration of rationals: Recall our code for enumerating rationals using `ArrayList[E]`.

```
# Output reduced positive fractions, i.e., positive rationals.
reduced = ArrayList[Rational]()
d = 0
while True:
    r = d
    for c in range(0, d+1):
        # Let z be the reduced form of the fraction (r+1)/(c+1)
        z: Rational = Rational(r + 1, c + 1)

        # On first occurrence of z, print it and add to reduced.
        if not reduced.contains(z):
            print(z)
            reduced.add(z)

        # Move diagonally up and (by virtue of c's increment) to the right.
        r -= 1
    d += 1
```

Enumeration of rationals: To use `HashSet[E]` instead, we only need to change one word.

```
# Output reduced positive fractions, i.e., positive rationals.
reduced = HashSet[Rational]()
d = 0
while True:
    r = d
    for c in range(0, d+1):
        # Let z be the reduced form of the fraction (r+1)/(c+1)
        z: Rational = Rational(r + 1, c + 1)

        # On first occurrence of z, print it and add to reduced.
        if not reduced.contains(z):
            print(z)
            reduced.add(z)

        # Move diagonally up and (by virtue of c's increment) to the right.
        r -= 1
    d += 1
```

The application code invoking contains and add is unchanged, but the methods that are dynamically dispatched change radically, i.e., from the `ArrayList[E]` implementations to the `HashSet[E]` implementations.

Enumeration of rationals: To use `HashSet[E]` instead, we only need to change one line.

```
# Output reduced positive fractions, i.e., positive rationals.
reduced = HashSet[Rational]()
d = 0
while True:
    r = d
    for c in range(0, d+1):
        # Let z be the reduced form of the fraction (r+1)/(c+1)
        z: Rational = Rational(r + 1, c + 1)

        # On first occurrence of z, print it and add to reduced.
        if not reduced.contains(z):
            print(z)
            reduced.add(z)

        # Move diagonally up and (by virtue of c's increment) to the right.
        r -= 1
    d += 1
```

Comment out the print statement so that printing does not mask timing differences. Then, emit elapsed time every 10,000 insertions into reduced.

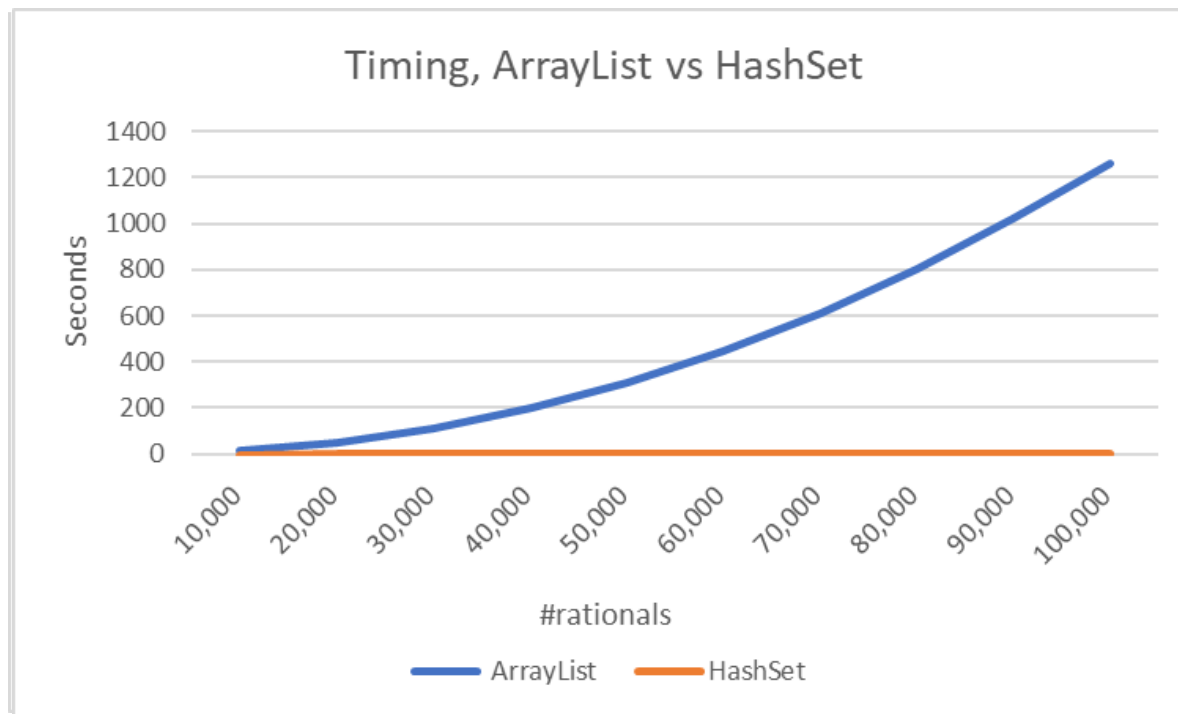
Timing Study: Contrast performance of `ArrayList` and `HashSet`.

```
# Output reduced positive fractions, i.e., positive rationals.
reduced = ArrayList[Rational](20)
startTime = time.time();
count = 0; # number of rationals so far.
d = 0
while count < 100000:
    r = d
    for c in range(0, d+1):
        # Let z be the reduced form of the fraction (r+1)/(c+1)
        z: Rational = Rational(r + 1, c + 1)

        # On first occurrence of z, print it and add to reduced.
        if not reduced.contains(z):
            # print(z)
            reduced.add(z)
            count += 1
            if (count % 10000) == 0: print(time.time() - startTime);

        # Move diagonally up and (by virtue of c's increment) to the right.
        r -= 1
    d += 1
```


Timing Study: Contrast performance of ArrayList and HashSet.



The running time of ArrayList is quadratic, whereas the running time of HashSet is linear and negligible.

Timing Study: But why are we bothering to maintain the collection of already-output rationals in the first place? We thought this was needed to make sure we would only emit each rational once.

But why not just output the fractions that are in reduced form as they arise? We didn't actually need the collection in the first place. It was all just a **pedagogical ruse!**

The test for n/d being in reduced form is " $\text{gcd}(n,d)==1$ ".



Analyze first.

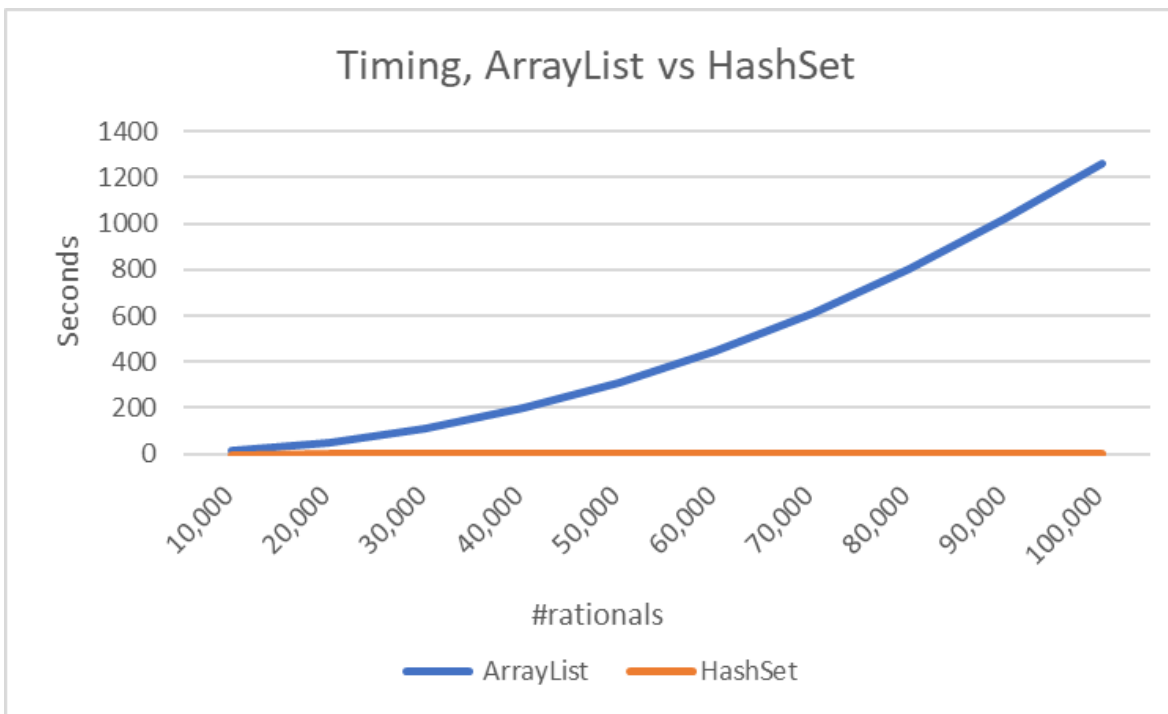
We make the Rational just to keep the timings comparable.

Enumeration of rationals: Contrast performance of ArrayList, HashSet, and gcd=1.

```
# Output reduced positive fractions, i.e., positive rationals.
startTime = time.time();
count = 0; # number of rationals so far.
d = 0
while count < 100000:
    r = d
    for c in range(0, d+1):
        # Make rational from (r+1)/(c+1) and count it if it's in reduced form.
        if gcd(r + 1, c + 1) == 1:
            z: Rational = Rational(r + 1, c + 1)
            # print(z)
            count += 1
            if (count % 10000) == 0: print(time.time() - startTime);

        # Move diagonally up and (by virtue of c's increment) to the right.
        r -= 1
    d += 1
```

Enumeration of rationals: Contrast performance of ArrayList, HashSet, and gcd=1.



#rationals	ArrayList	HashSet	gcd
10,000	12	0.11	0.02
20,000	47	0.32	0.05
30,000	109	0.63	0.09
40,000	196	0.92	0.12
50,000	310	1.24	0.15
60,000	448	1.69	0.19
70,000	612	2.07	0.22
80,000	802	2.49	0.25
90,000	1020	2.93	0.29
100,000	1263	3.46	0.33

The running time of ArrayList is quadratic, whereas the running time of HashSet is linear and negligible. But in comparison, checking whether the fraction is in reduced form is practically instantaneous.

Enumerating a Collection: A small loose end.

Recall (from Chapter 12) that one of the operations of a collection is to enumerate its elements. This is easy when we have **direct access** to the collection's implementation, e.g.,

```
# Enumerate items of a collection implemented as a list (A[0:size-1], size, n).  
for k in range(0, size): # Do whatever for A[k].
```

```
# Enumerate items of a collection implemented as a histogram H[0..maxValue].  
for k in range(0, size):  
    for j in range(0, H[k]):  
        # Do whatever for k.
```

But how can you enumerate the items of a collection when its implementation is hidden within a class? What is needed is **indirect access** to the collection's implementation.

Enumerating a Collection: A small loose end.

Specifically, we want to write client code that is independent of the collection's implementation, e.g.

```
collection = _____  
...  
collection.add(2)  
collection.add(3)  
collection.add(5)  
collection.add(7)  
...  
for k in collection: print(k)
```

and have it work regardless of how the collection is implemented, e.g., as either of the following:

```
collection = ArrayList[int]()           or           collection = HashSet[int]()
```

Enumerating a Collection: A small loose end.

What is needed is the notion of an *iterator*, an object that can be “pumped” to obtain successive elements of the collection until, when exhausted, it raises an exception. In Python, the method of a class that is invoked to yield a new iterator is `__iter__()`, and the method of an iterator that provides successive elements is `__next__()`. An object can serve as its own iterator.

ArrayList[E]

```
def __iter__(self):
    self._index = 0
    return self
def __next__(self):
    if self._index < self._size:
        value = self._A[self._index]
        self._index += 1
        return value
    else:
        raise StopIteration
```

HashSet[E]

```
def __iter__(self):
    self._iterator = iter(self._A)
    return self
def __next__(self):
    return next(self._iterator)
```

A new iterator for an ArrayList sets the instance variable `_index` to 0.

Enumerating a Collection: A small loose end.

What is needed is the notion of an *iterator*, an object that can be “pumped” to obtain successive elements of the collection until, when exhausted, it raises an exception. In Python, the method of a class that is invoked to yield a new iterator is `__iter__()`, and the method of an iterator that provides successive elements is `__next__()`. An object can serve as its own iterator.

ArrayList[E]

```
def __iter__(self):
    self._index = 0
    return self
def __next__(self):
    if self._index < self._size:
        value = self._A[self._index]
        self._index += 1
        return value
    else:
        raise StopIteration
```

HashSet[E]

```
def __iter__(self):
    self._iterator = iter(self._A)
    return self
def __next__(self):
    return next(self._iterator)
```


Successive elements of the `ArrayList` are obtained by subscripting `_A` and then incrementing `_index` until reaching `_size`, at which point the `StopIteration` exception is raised.

Enumerating a Collection: A small loose end.

What is needed is the notion of an *iterator*, an object that can be “pumped” to obtain successive elements of the collection until, when exhausted, it raises an exception. In Python, the method of a class that is invoked to yield a new iterator is `__iter__()`, and the method of an iterator that provides successive elements is `__next__()`. An object can serve as its own iterator.

`ArrayList[E]`

```
def __iter__(self):
    self._index = 0
    return self
def __next__(self):
    if self._index < self._size:
        value = self._A[self._index]
        self._index += 1
        return value
    else:
        raise StopIteration
```

`HashSet[E]`

```
def __iter__(self):
    self._iterator = iter(self._A)
    return self
def __next__(self):
    return next(self._iterator)
```

A new iterator for a HashSet just gets a new iterator from the underlying set implementation.

Enumerating a Collection: A small loose end.

What is needed is the notion of an *iterator*, an object that can be “pumped” to obtain successive elements of the collection until, when exhausted, it raises an exception. In Python, the method of a class that is invoked to yield a new iterator is `__iter__()`, and the method of an iterator that provides successive elements is `__next__()`. An object can serve as its own iterator.

ArrayList[E]

```
def __iter__(self):
    self._index = 0
    return self
def __next__(self):
    if self._index < self._size:
        value = self._A[self._index]
        self._index += 1
        return value
    else:
        raise StopIteration
```

HashSet[E]

```
def __iter__(self):
    self._iterator = iter(self._A)
    return self
def __next__(self):
    return next(self._iterator)
```

Successive elements of the `HashSet` are obtained by invoking the `next` operation of the underlying set implementation until it raises the `StopIteration` exception.

Enumerating a Collection: A small loose end.

What is needed is the notion of an *iterator*, an object that can be “pumped” to obtain successive elements of the collection until, when exhausted, it raises an exception. In Python, the method of a class that is invoked to yield a new iterator is `__iter__()`, and the method of an iterator that provides successive elements is `__next__()`. An object can serve as its own iterator.

`ArrayList[E]`

```
def __iter__(self):
    self._index = 0
    return self
def __next__(self):
    if self._index < self._size:
        value = self._A[self._index]
        self._index += 1
        return value
    else:
        raise StopIteration
```

`HashSet[E]`

```
def __iter__(self):
    self._iterator = iter(self._A)
    return self
def __next__(self):
    return next(self._iterator)
```

Summary:

We have presented the flavor of Object-Oriented Programming (OOP), and some of its technical details.

We reinforced many of the lessons of earlier chapters:

- Combining careful specification of statements, declarations, and methods with careful implementations.
- The practice of incremental testing.
- The benefit of analysis.

Object-Oriented Programming addresses for code many issues that scholars have considered in Philosophy:

- The nature of taxonomy.
- Abstraction and instantiation.
- Ontology and epistemology.