Composition, Representation
Class outline:

- Composition
- Representation
Composition
Composition

An object can contain references to objects of other classes.

What examples of composition are in an animal conservatory?

• An animal has a mate.
• An animal has a mother.
• An animal has children.
• A conservatory has animals.
Referencing other instances

An instance variable can refer to another instance.

We can add this method to the base Animal class that adds a mate instance variable:

class Animal:

    def mate_with(self, other):
        if other is not self and other.species_name == self.species_name:
            self.mate = other
            other.mate = self

How would we call that method?
Referencing other instances

An instance variable can refer to another instance.

We can add this method to the base Animal class that adds a **mate** instance variable:

```python
class Animal:
    def mate_with(self, other):
        if other is not self and other.species_name == self.species_name:
            self.mate = other
            other.mate = self
```

How would we call that method?

```python
mr_wabbit = Rabbit("Mister Wabbit", 3)
jane_doe = Rabbit("Jane Doe", 2)
mr_wabbit.mate_with(jane_doe)
```
Referencing a list of instances

An instance variable can also store a list of instances.

We can add this method to the Rabbit class that adds a babies instance variable.

class Rabbit(Animal):

    def reproduce_like_rabbits(self):
        if self.mate is None:
            print("oh no! better go on ZoOkCupid")
        return
        self.babies = []
        for _ in range(0, self.num_in_litter):
            self.babies.append(Rabbit("bunny", 0))

How would we call that function?
Referencing a list of instances

An instance variable can also store a list of instances.

We can add this method to the Rabbit class that adds a babies instance variable.

class Rabbit(Animal):
    def reproduce_like_rabbits(self):
        if self.mate is None:
            print("oh no! better go on ZoOkCupid")
        return
        self.babies = []
        for _ in range(0, self.num_in_litter):
            self.babies.append(Rabbit("bunny", 0))

How would we call that function?

mr_wabbit = Rabbit("Mister Wabbit", 3)
jane_doe = Rabbit("Jane Doe", 2)
mr_wabbit.mate_with(jane_doe)
jane_doe.reproduce_like_rabbits()
Relying on a common interface

If all instances implement a method with the same function signature, a program can rely on that method across instances of different subclasses.

```python
def partytime(animals):
    """Assuming ANIMALS is a list of Animals, cause each to interact with all the others exactly once.""
    for i in range(len(animals)):
        for j in range(i + 1, len(animals)):
            animals[i].interact_with(animals[j])
```

How would we call that function?
Relying on a common interface

If all instances implement a method with the same function signature, a program can rely on that method across instances of different subclasses.

```python
def partytime(animals):
    '''Assuming ANIMALS is a list of Animals, cause each
    to interact with all the others exactly once.'''
    for i in range(len(animals)):
        for j in range(i + 1, len(animals)):
            animals[i].interact_with(animals[j])
```

How would we call that function?

```python
jane_doe = Rabbit("Jane Doe", 2)
scar = Lion("Scar", 12)
elly = Elephant("Elly", 5)
pandy = Panda("PandeyBear", 4)
partytime([jane_doe, scar, elly, pandy])
```
Composition vs. Inheritance

Inheritance is best for representing "is-a" relationships

- Rabbit is a specific type of Animal
- So, Rabbit inherits from Animal

Composition is best for representing "has-a" relationships

- A conservatory has a collection of animals it cares for
- So, a conservatory has a list of animals as an instance variable
Objects everywhere
class Lamb:
    species_name = "Lamb"
    scientific_name = "Ovis aries"

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

    def play(self):
        self.happy = True

lamb = Lamb("Lil")
owner = "Mary"
had_a_lamb = True
fleece = {"color": "white", "fluffiness": 100}
kids_at_school = ["Billy", "Tilly", "Jilly"]
day = 1
So many objects

What are the objects in this code?

class Lamb:
    species_name = "Lamb"
    scientific_name = "Ovis aries"

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

    def play(self):
        self.happy = True

lamb = Lamb("Lil")
owner = "Mary"
had_a_lamb = True
fleece = {"color": "white", "fluffiness": 100}
kids_at_school = ["Billy", "Tilly", "Jilly"]
day = 1

lamb, owner, had_a_lamb, fleece, kids_at_school, day, etc. We can prove it by checking object.__class__.__bases__, which reports the base class(es) of the object's class.
It's all objects

All the built-in types inherit from \texttt{object}:
Built-in object attributes

If all the built-in types and user classes inherit from \texttt{object}, what are they inheriting?

Just ask \texttt{dir()}, a built-in function that returns a list of all the "interesting" attributes on an object.

\begin{verbatim}
dir(object)
\end{verbatim}
Built-in object attributes

If all the built-in types and user classes inherit from `object`, what are they inheriting?

Just ask `dir()`, a built-in function that returns a list of all the "interesting" attributes on an object.

```
   dir(object)
```

- For string representation: `__repr__`, `__str__`, `__format__`
- For comparisons: `__eq__`, `__ge__`, `__gt__`, `__le__`, `__lt__`, `__ne__`
- Related to classes: `__bases__`, `__class__`, `__new__`, `__init__`, `__init_subclass__`, `__subclasshook__`, `__setattr__`, `__delattr__`, `__getattribute__`
- Others: `__dir__`, `__hash__`, `__module__`, `__reduce__`, `__reduce_ex__`

Python calls these methods behind these scenes, so we are often not aware when the "dunder" methods are being called. Let us become enlightened!
String representation
**__str__**

The **__str__** method returns a human readable string representation of an object.

```python
from fractions import Fraction

one_third = 1/3
one_half = Fraction(1, 2)

float.__str__(one_third)
Fraction.__str__(one_half)
```
__str__

The __str__ method returns a human readable string representation of an object.

```python
from fractions import Fraction

one_third = 1/3
one_half = Fraction(1, 2)

float.__str__(one_third)  # '0.33333333333333333333333333333333'
Fraction.__str__(one_half)  # '1/2'
```
The `__str__` method is used in multiple places by Python: `print()` function, `str()` constructor, f-strings, and more.

```python
from fractions import Fraction

one_third = 1/3
one_half = Fraction(1, 2)

print(one_third)
print(one_half)

str(one_third)
str(one_half)

f"{one_half} > {one_third}"
__str__ usage

The __str__ method is used in multiple places by Python: `print()` function, `str()` constructor, f-strings, and more.

```python
from fractions import Fraction

one_third = 1/3
one_half = Fraction(1, 2)

print(one_third)  # '0.3333333333333333'
print(one_half)   # '1/2'

str(one_third)    # '0.3333333333333333'
str(one_half)     # '1/2'

f"{one_half} > {one_third}"  # '1/2 > 0.3333333333333333'
```
Custom __str__ behavior

When making custom classes, we can override __str__ to define our human readable string representation.

class Lamb:
    species_name = "Lamb"
    scientific_name = "Ovis aries"

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

    def __str__(self):
        return "Lamb named " + self.name

lil = Lamb("Lil lamb")

str(lil)

print(lil)
The `__repr__` method returns a string that would evaluate to an object with the same values.

```python
from fractions import Fraction
one_half = Fraction(1, 2)
Fraction.__repr__(one_half) # 'Fraction(1, 2)'
```

If implemented correctly, calling `eval()` on the result should return back that same-valued object.

```python
another_half = eval(Fraction.__repr__(one_half))
```
__repr__ usage

The __repr__ method is used multiple places by Python: when `repr(object)` is called and when displaying an object in an interactive Python session.

```python
from fractions import Fraction

one_third = 1/3
one_half = Fraction(1, 2)

one_third
one_half
repr(one_third)
repr(one_half)
```
Custom __repr__ behavior

When making custom classes, we can override __repr__ to return a more appropriate Python representation.

class Lamb:
    species_name = "Lamb"
    scientific_name = "Ovis aries"

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

    def __str__(self):
        return "Lamb named " + self.name

    def __repr__(self):
        return f"Lamb({repr(self.name})"

lil = Lamb("Lil lamb")
repr(lil)
lil
The rules of `repr` and `str`

When the `repr(obj)` function is called:

- Python calls the `ClassName.__repr__` method if it exists.
- If `ClassName.__repr__` does not exist, Python will look up the chain of parent classes until it finds one with `__repr__` defined.
- If all else fails, `object.__repr__` will be called.

When the `str(obj)` class constructor is called:

- Python calls the `ClassName.__str__` method if it exists.
- If no `__str__` method is found on that class, Python calls `repr()` on the object instead.
- ↑ See above!
Special methods
Special methods

Special methods have built-in behavior. Special method names always start and end with double underscores.

<table>
<thead>
<tr>
<th>Name</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>init</code></td>
<td>Method invoked automatically when an object is constructed</td>
</tr>
<tr>
<td><code>repr</code></td>
<td>Method invoked to display an object as a Python expression</td>
</tr>
<tr>
<td><code>str</code></td>
<td>Method invoked to stringify an object</td>
</tr>
<tr>
<td><code>add</code></td>
<td>Method invoked to add one object to another</td>
</tr>
<tr>
<td><code>bool</code></td>
<td>Method invoked to convert an object to True or False</td>
</tr>
<tr>
<td><code>float</code></td>
<td>Method invoked to convert an object to a float (real number)</td>
</tr>
</tbody>
</table>

See all special method names.
# Special method examples

```python
zero = 0
one = 1
two = 2
```

<table>
<thead>
<tr>
<th><strong>Standard approach</strong></th>
<th><strong>Dunder equivalent</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>one + two</code> # 3</td>
<td><code>one.__add__(two)</code> # 3</td>
</tr>
<tr>
<td><code>bool(zero)</code> # False</td>
<td><code>zero.__bool__()</code> # False</td>
</tr>
<tr>
<td><code>bool(one)</code> # True</td>
<td><code>one.__bool__()</code> # True</td>
</tr>
</tbody>
</table>
Adding together custom objects

Consider the following class:

```python
from math import gcd

class Rational:
    def __init__(self, numerator, denominator):
        g = gcd(numerator, denominator)
        self.numer = numerator // g
        self.denom = denominator // g

    def __str__(self):
        return f"{self.numer}/{self.denom}"

    def __repr__(self):
        return f"Rational({self.numer}, {self.denom})"

Rational(1, 2) + Rational(3, 4)
```

Will this work?
Adding together custom objects

Consider the following class:

```python
from math import gcd

class Rational:
    def __init__(self, numerator, denominator):
        g = gcd(numerator, denominator)
        self.numer = numerator // g
        self.denom = denominator // g

    def __str__(self):
        return f"{self.numer}/{self.denom}"

    def __repr__(self):
        return f"Rational({self.numer}, {self.denom})"

Rational(1, 2) + Rational(3, 4)
```

Will this work?

```python
Rational(1, 2) + Rational(3, 4)
```

TypeError: unsupported operand type(s) for +: 'Rational' and 'Rational'
Implementing dunder methods

We can make instances of custom classes addable by defining the `__add__` method:

class Rational:
    def __init__(self, numerator, denominator):
        g = gcd(numerator, denominator)
        self.numer = numerator // g
        self.denom = denominator // g

    def __add__(self, other):

    # The rest...
Implementing dunder methods

We can make instances of custom classes addable by defining the `__add__` method:

class Rational:
    def __init__(self, numerator, denominator):
        g = gcd(numerator, denominator)
        self.numer = numerator // g
        self.denom = denominator // g

    def __add__(self, other):
        new_numer = self.numer * other.denom + other.numer * self.denom
        new_denom = self.denom * other.denom
        return Rational(new_numer, new_denom)

    # The rest...
Implementing dunder methods

We can make instances of custom classes addable by defining the `__add__` method:

class Rational:
    def __init__(self, numerator, denominator):
        g = gcd(numerator, denominator)
        self.numer = numerator // g
        self.denom = denominator // g

    def __add__(self, other):
        new_numer = self.numer * other.denom + other.numer * self.denom
        new_denom = self.denom * other.denom
        return Rational(new_numer, new_denom)

# The rest...

Now try...

Rational(1, 2) + Rational(3, 4)
Polymorphism
Polymorphic functions

**Polymorphic function**: A function that applies to many (poly) different forms (morph) of data.

`str` and `repr` are both polymorphic; they apply to any object.

```python
repr(1/3)  # '0.3333333333333333'
repr(Rational(1, 3))  # 'Rational(1, 3)'

str(1/3)  # '0.3333333333333333'
str(Rational(1, 3))  # '1/3'
```

The class of that object can customize the per-object behavior using `__str__` and `__repr__`. 
Generic functions

A **generic function** can apply to arguments of different types.

```python
def sum_two(a, b):
    return a + b
```

What could $a$ and $b$ be?

The function `sum_two` is **generic** in the type of $a$ and $b$. 
Generic functions

A **generic function** can apply to arguments of different types.

```python
def sum_two(a, b):
    return a + b
```

What could \(a\) and \(b\) be? Anything summable!

The function `sum_two` is **generic** in the type of \(a\) and \(b\).
Generic function #2

```python
def sum_em(items, initial_value):
    """Returns the sum of ITEMS, starting with a value of INITIAL_VALUE.""
    sum = initial_value
    for item in items:
        sum += item
    return sum
```

What could `items` be?

What could `initial_value` be?

The function `sum_em` is generic in the type of `items` and the type of `initial_value`. 
Generic function #2

```python
def sum_em(items, initial_value):
    '''Returns the sum of ITEMS, starting with a value of INITIAL_VALUE.'''
    sum = initial_value
    for item in items:
        sum += item
    return sum
```

What could `items` be? Any iterable with summable values.

What could `initial_value` be?

The function `sum_em` is generic in the type of `items` and the type of `initial_value`. 
Generic function #2

```python
def sum_em(items, initial_value):
    """Returns the sum of ITEMS, starting with a value of INITIAL_VALUE."""
    sum = initial_value
    for item in items:
        sum += item
    return sum
```

What could `items` be? Any iterable with summable values.

What could `initial_value` be? Any value that can be summed with the values in iterable.

The function `sum_em` is generic in the type of `items` and the type of `initial_value`.
Type dispatching

Another way to make generic functions is to select a behavior based on the type of the argument.

```python
def is_valid_month(month):
    if isinstance(month, int):
        return month >= 1 and month <= 12
    elif isinstance(month, str):
        return month in ["January", "February", "March", "April",
                         "May", "June", "July", "August", "September",
                         "October", "November", "December"]
    return False
```

What could `month` be?

The function `is_valid_month` is generic in the type of `month`. 
Type dispatching

Another way to make generic functions is to select a behavior based on the type of the argument.

```python
def is_valid_month(month):
    if isinstance(month, int):
        return month >= 1 and month <= 12
    elif isinstance(month, str):
        return month in ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December']
    return False
```

What could `month` be? Either an int or string.

The function `is_valid_month` is **generic** in the type of `month`. 
Type coercion

Another way to make generic functions is to coerce an argument into the desired type.

```python
def sum_numbers(nums):
    """Returns the sum of NUMS""
    sum = Rational(0, 0)
    for num in nums:
        if isinstance(num, int):
            num = Rational(num, 1)
        sum += num
    return sum
```

What could `nums` be?

The function `sum_numbers` is **generic** in the type of `nums`.
Type coercion

Another way to make generic functions is to coerce an argument into the desired type.

```python
def sum_numbers(nums):
    """Returns the sum of NUMS""
    sum = Rational(0, 0)
    for num in nums:
        if isinstance(num, int):
            num = Rational(num, 1)
        sum += num
    return sum
```

What could `nums` be? Any iterable with ints or Rationals.

The function `sum_numbers` is generic in the type of `nums`.