Special Object Methods

(Revisit) Composition

A composition challenge

Composition: When one object is composed of another object(s).

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

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

class Human:
    species_name = "Human"
    scientific_name = "Homo sapiens"

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

lamb = Lamb("little")
mary = Human("Mary")
```

How can we make it so that Mary has a little lamb?

Approach 1: Assign object in method

Without changing the `__init__`, we can add a method that assigns a new instance variable.

```
class Human:
    species_name = "Human"
    scientific_name = "Homo sapiens"

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

    def adopt(self, pet):
        self.pet = pet
        print(f"I have a pet named {self.pet.name}")

lamb = Lamb("little")
mary = Human("Mary")
mary.adopt(lamb)
print(mary.pet)
```

What will happen if we call `mary.pet` before `mary.adopt(pet)`?

Approach 1: Assign object in method

Without changing the `__init__`, we can add a method that assigns a new instance variable.

```
class Human:
    species_name = "Human"
    scientific_name = "Homo sapiens"

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

    def adopt(self, pet):
        self.pet = pet
        print(f"I have a pet named {self.pet.name}")

lamb = Lamb("little")
mary = Human("Mary")
mary.adopt(lamb)
print(mary.pet)
```

What will happen if we call `mary.pet` before `mary.adopt(pet)`? ❌ AttributeError!

Approach 2: Assign during initialization

We can change `__init__` to accept the object as an argument and initialize the instance variable immediately.

```
class Human:
    species_name = "Human"
    scientific_name = "Homo sapiens"

    def __init__(self, name, pet=None):
        self.name = name
        self.pet = pet
        print(f"I have a pet named {self.pet.name}")

lamb = Lamb("little")
mary = Human("Mary", lamb)
```

How would you construct a `Human` that has no pet?
What will their `pet` attribute be?

Approach 2: Assign during initialization

We can change `__init__` to accept the object as an argument and initialize the instance variable immediately.

```
class Human:
    species_name = "Human"
    scientific_name = "Homo sapiens"

    def __init__(self, name, pet=None):
        self.name = name
        self.pet = pet
        print(f"I have a pet named {self.pet.name}")

lamb = Lamb("little")
mary = Human("Mary", lamb)
```

How would you construct a `Human` that has no pet? `Human("Colby")`
What will their `pet` attribute be? `None`

Approach 3: Update a list

We can initialize an empty list in `__init__` and use a method to update the list.

```
class Human:
    species_name = "Human"
    scientific_name = "Homo sapiens"

    def __init__(self, name):
        self.name = name
        self.pets = []

    def adopt(self, pet):
        self.pets.append(pet)
        print(f"I have a pet named {pet.name}")

lamb = Lamb("little")
mary = Human("Mary")
mary.adopt(lamb)
```

What method would be useful to add to this class?

Approach 3: Update a list

We can initialize an empty list in `__init__` and use a method to update the list.

```
class Human:
    species_name = "Human"
    scientific_name = "Homo sapiens"

    def __init__(self, name):
        self.name = name
        self.pets = []

    def adopt(self, pet):
        self.pets.append(pet)
        print(f"I have a pet named {pet.name}")

lamb = Lamb("little")
mary = Human("Mary")
mary.adopt(lamb)
```

What method would be useful to add to this class? Something to remove a pet, in case the pet runs away or something happens...

Objects

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
```

So many objects

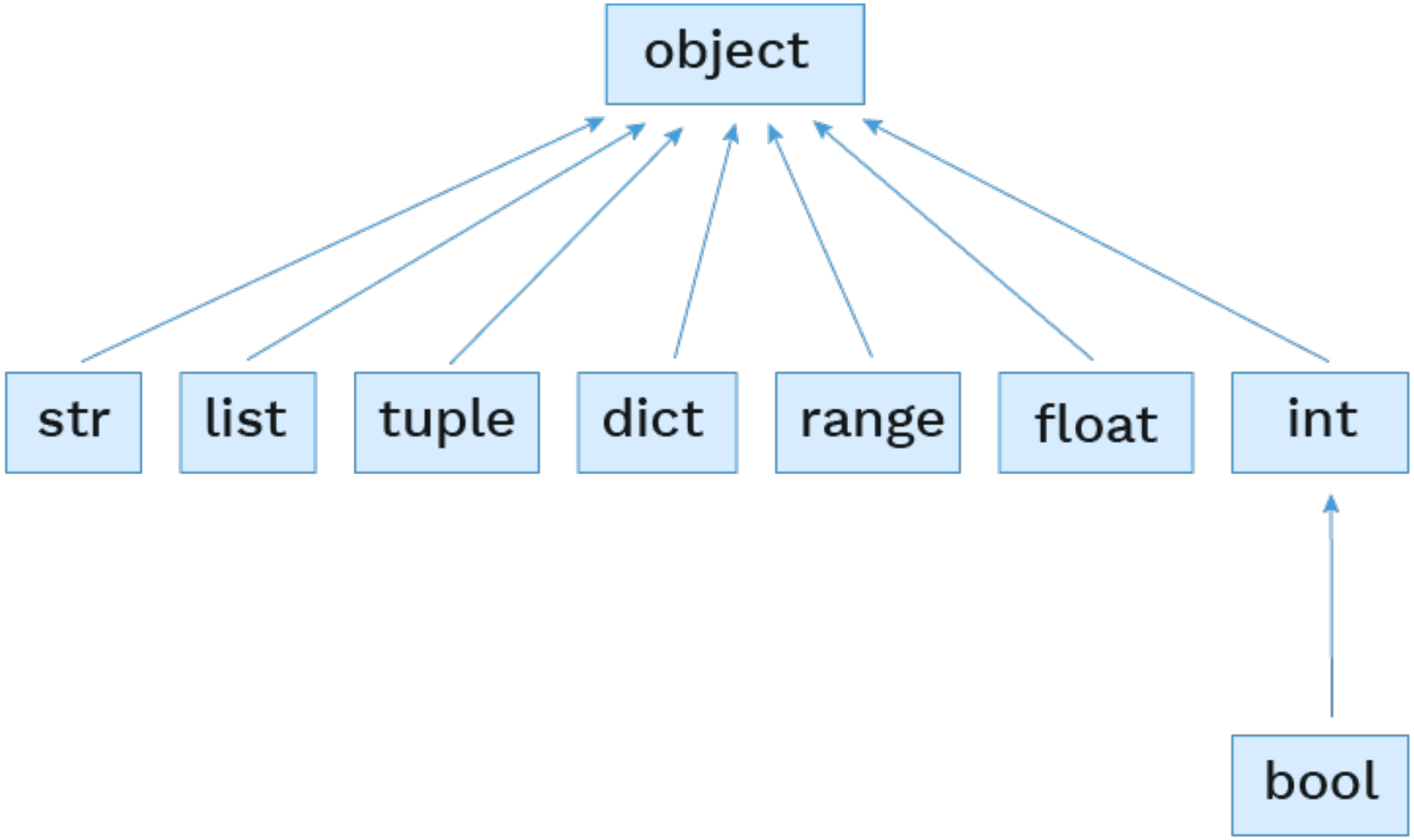
What are the objects in this code?

```
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 `object`:



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 attributes on an object.

```
dir(object)
```

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 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.

```
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.

```
from fractions import Fraction

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

```
float.__str__(one_third)      # '0.3333333333333333'
Fraction.__str__(one_half)    # '1/2'
```

__str__ usage

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

```
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.

```
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 "🐑 : " + self.name
```

```
lil = Lamb("Lil lamb")

str(lil)

print(lil) # Currently broken on code.cs61a.org!
```

__repr__

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

```
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.

```
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.

```
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 "🐑 : " + self.name

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

```
lil = Lamb("Lil lamb")
repr(lil)
lil
```

Attribute access

Get attribute with dot notation

`expression.attribute` evaluates to the value of `attribute` in the object referenced by `expression`.

```
class Bunny:
    species_name = "Bunny"
    scientific_name = "Bunnius Bunalot"

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

```
bunny = Bunny("Boo")
bunny.name
bunny.species_name
bunny.scientific_name
bunny.ears_hang_low
bunny.tie_ears()
```

Get attribute with dot notation

`expression.attribute` evaluates to the value of `attribute` in the object referenced by `expression`.

```
class Bunny:
    species_name = "Bunny"
    scientific_name = "Bunnius Bunalot"

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

```
bunny = Bunny("Boo")
bunny.name
bunny.species_name
bunny.scientific_name
bunny.ears_hang_low      # ❌ AttributeError!
bunny.tie_ears()         # ❌ AttributeError!
```

Python raises an exception if you try to access an attribute that does not exist.

Get attribute with getattr()

`getattr(object, name[, default])` looks up the attribute by `name` on `object`.

If it's undefined, it returns `default` if provided or raises `AttributeError` otherwise.

```
bunny = Bunny("Bugs")
getattr(bunny, "ears_hang_low")
getattr(bunny, "ears_hang_low", False)
getattr(bunny, "tie_ears")
getattr(bunny, "tie_ears",
                lambda self: print("ears tied!"))
```

Get attribute with getattr()

`getattr(object, name[, default])` looks up the attribute by `name` on `object`.

If it's undefined, it returns `default` if provided or raises `AttributeError` otherwise.

```
bunny = Bunny("Bugs")
getattr(bunny, "ears_hang_low")           # ❌ AttributeError!
getattr(bunny, "ears_hang_low", False)    # False
getattr(bunny, "tie_ears")                # ❌ AttributeError!
getattr(bunny, "tie_ears",                # lambda
                lambda self: print("ears tied!"))
```

Behind the scenes: __getattribute__

When we call `object.name` or `getattr(object, name)`, Python calls `__getattribute__` on the object.

```
class Light(object):

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

    def __getattribute__(self, name):
        print('__getattribute__ ', name)
        return super().__getattribute__(name)
```

```
lamp = Light(750)
lamp.brightness
getattr(lamp, "brightness")
Light.__getattribute__(lamp, "brightness")
```

Check attribute exists with dot notation

```
class Bunny:
    species_name = "Bunny"
    scientific_name = "Bunnius Bunalot"

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

```
bunny = Bunny("Shelby")
if bunny.ears_hang_low:
    print("Yes my ears hang low, they wobble to and fro")
else:
    print("Alas, I am not a lop!")
```

What will happen?

Check attribute exists with dot notation

```
class Bunny:
    species_name = "Bunny"
    scientific_name = "Bunnius Bunalot"

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

```
bunny = Bunny("Shelby")
if bunny.ears_hang_low:
    print("Yes my ears hang low, they wobble to and fro")
else:
    print("Alas, I am not a lop!")
```

What will happen?

❌AttributeError! Python raises an exception if you try to access an attribute that does not exist.

Check attribute exists with hasattr()

`hasattr(object, name)` looks up the attribute by `name` on `object` and returns whether it can find such an attribute.

```
class Bunny:
    species_name = "Bunny"
    scientific_name = "Bunnius Bunalot"

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

```
bunny = Bunny("Colby")
if hasattr(bunny, "ears_hang_low"):
    print("Yes my ears hang low, they wobble to and fro")
else:
    print("Alas, I am not a lop!")
```

Python implements this function by calling `getattr()` and checking to see if an exception is returned, so this function also ends up calling `__getattribute__`.

There's more!

Special methods

Here are more special methods on objects:

Method	Implements
`__setattr__(obj, "n", v)`	`x.n = v`
`__delattr__(obj, "n")`	`del x.n`
`__eq__(obj, x)`	`obj == x`
`__ne__(obj, x)`	`obj != x`
`__ge__(obj, x)`	`obj >= x`
`__gt__(obj, x)`	`obj > x`
`__le__(obj, x)`	`obj <= x`
`__lt__(obj, x)`	`obj < x`

That's not all! There are many more special method names that you can define on objects to customize how Python operates on them.