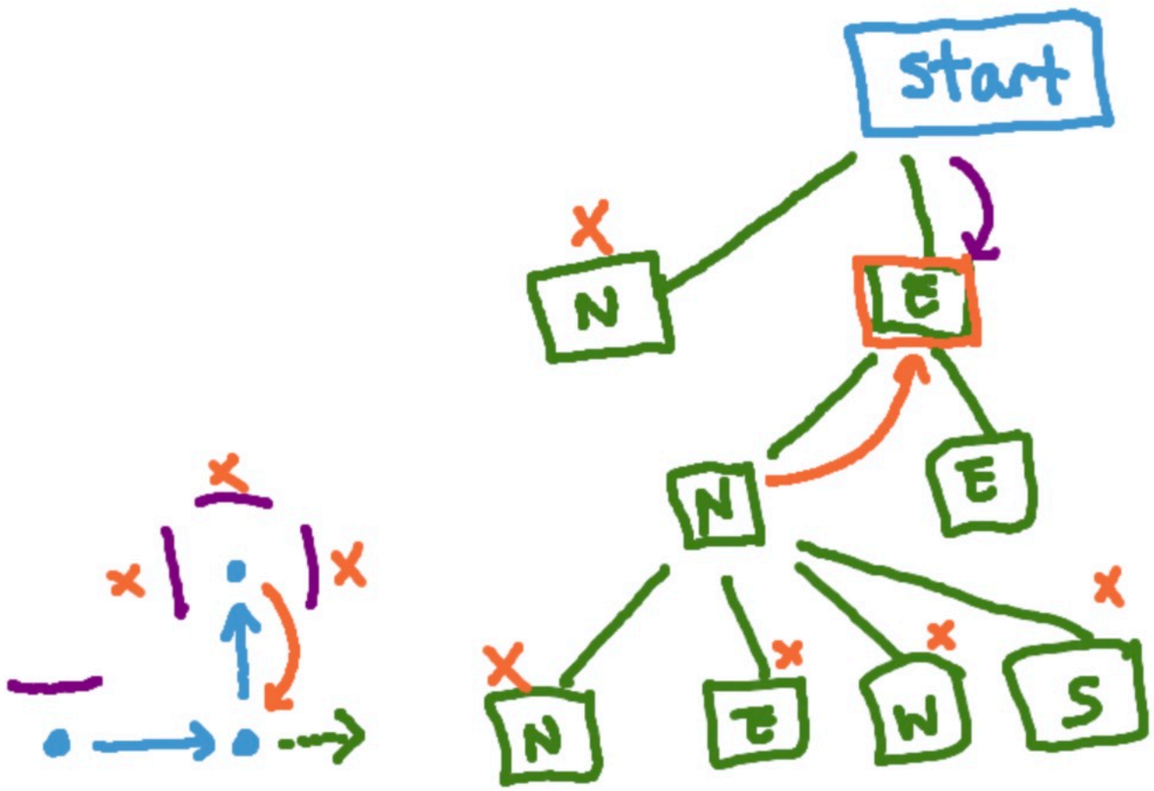
Middlebury

CSCI 146: Intensive Introduction to Computing

Fall 2025

Lecture 14: Object-Oriented Programming

A structured way to approach the maze problem.



CS 202

CS 200 — math

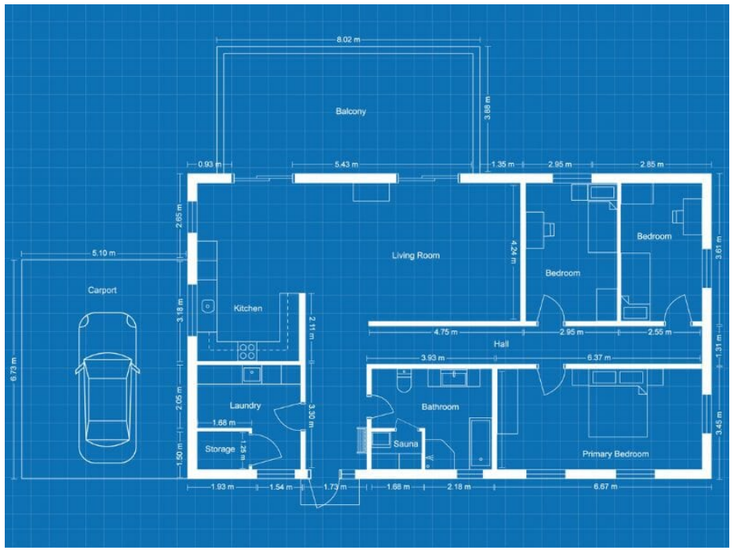
CS 201 — implementing

Goals for today

- Explain the relationship between types, classes and objects in Python
- Implement a class that overrides operators
- Use inheritance to create a specialized types while reusing code

Terminology is important: look out for what each term means.

`Class` $\longleftrightarrow$ blueprint for a house,
`object` (or class instance) $\longleftrightarrow$ house,
`reference` $\longleftrightarrow$ address of house.

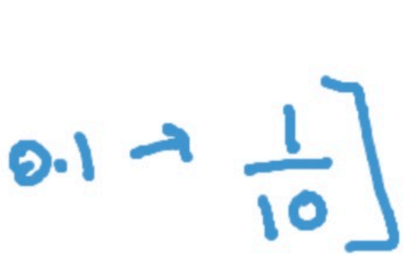


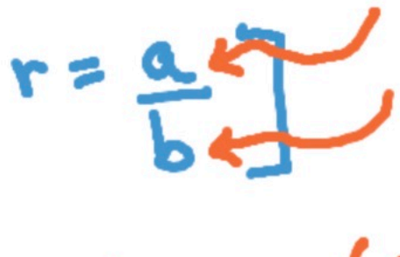
Motivating example: representing rational numbers.

```
>>> 0.1 + 0.2 <= 0.3
```

```
>>> import decimal
>>> decimal.Decimal(0.1 + 0.2)
>>> decimal.Decimal(0.3)
```

It would be nice if we can make our own type to represent rational numbers. Ideas?





tuple (a, b)

$(a_1, b_1) + (a_2, b_2) \rightarrow \frac{a_1}{b_1} + \frac{a_2}{b_2}$

$= \frac{a_1 b_2 + a_2 b_1}{b_1 b_2} = \frac{a_3}{b_3}$

Representing rational numbers as a `tuple` makes it confusing to use. But! we can make our own `Rational` type by defining a `Rational` class:

```
class Rational:
    """Represent a rational number as the ratio of two integers

    Attributes:
        numerator, denominator: Integers defining this rational number
    """

    # Define an initializer that sets the numerator and denominator attributes.
    # It also needs a docstring, but note we don't include a return value since
    # it does not return. We also don't document the "self" parameter since it
    # already has a defined role in the Python language specification.
    def __init__(self, numerator, denominator):
        """
        Initialize a rational number from the numerator and denominator.

        Args:
            numerator, denominator: Integers defining this rational number
        """
        self.numerator = numerator
        self.denominator = denominator
```

Creating a new "instance of the `Rational` class" (i.e. object) and accessing attributes:

```
>>> r1 = Rational(1, 10)
>>> r1.numerator
1
>>> r1.denominator
10
>>> r2 = Rational(2, 10)
>>> r2.numerator
2
>>> r2.denominator
10
```

Let's define our own method to `add` two rational objects (which should also return a `Rational` object).

Question 1: Which of the following best describes the code elements below?

```
class Klass:
    def __init__(self, x):
        self.xcoord = x

    def act_on(self, value):
        self.xcoord += value

k = Klass(4)
```

A. `Klass` is a class, `xcoord` an instance variable, `act_on` is a method, `k` is an instance.

B. `Klass` is a class, `xcoord` a method, `act_on` is an instance variable, `k` is an instance.

C. `Klass` and `k` are instances, `xcoord` an instance variable, `act_on` is a method.

D. `Klass` and `k` are classes, `xcoord` an instance variable, `act_on` is a method.

E. `Klass` is a class, `xcoord` and `act_on` are methods, `act_on` is a method, `k` is an instance.

Question 2: After the above code executes, what is the value of `k.xcoord`?

```
class Klass:
    def __init__(self, x):
        self.xcoord = x

    def act_on(self, value):
        self.xcoord += value

k = Klass(4)
k.act_on(2)
```

A. "x"

B. 0

C. 2

D. 4

E. 6

"Dunder" methods (double underscore) methods are special methods that allow us to override operators.

Examples:

- `__add__` used for `+`
- `__eq__` used for `==`
- `__leq__` used for `<=`
- `__str__` used when we pass an object to the `print` function
- `__repr__` used to display variable (e.g. at the interpreter `>>>` prompt)

$$\frac{r_1 = a_1}{b_1} \equiv \frac{r_2 = a_2}{b_2} \quad ?$$

$$a_1 b_2 \equiv a_2 b_1$$

Question 3: Which correctly describes the methods on `Rational` executed in the code below?

```
r1 = Rational(1, 10)
r2 = Rational(2, 10)
r3 = r1 + r2
print(r2)
```

A. `__init__`

B. `__init__`, `__add__`

C. `__init__`, `__str__`

D. `__init__`, `__add__`, `__str__`

E. `__init__`, `__add__`, `__str__`, `__eq__`

Using "inheritance" to *derive* a *child* `Dollar` class from the *base* `Rational` class:

```
class Dollar(Rational):
    def __init__(self, cents):
        # Use `super()` to call `Rational`'s initializer,
        # passing through the cents as the numerator
        super().__init__(cents, 100)

    def __str__(self):
        dollars = self.numerator // 100
        cents = self.numerator % 100
        cents_str = str(cents) if cents >= 10 else "0" + str(cents)
        return "$" + str(dollars) + "." + cents_str
```

Summary and Reminders

- **Terminology:** class, object, method, instance variables, base/parent class, derived/child class.
- More OOP on Wednesday: application to games!
- Programming Assignment 5 final due date on Thursday.
- Programming Assignment 6 initial due date on Thursday.
- Use "Regrade Requests" form on the website. See Gradescope comments by clicking on **Code**.