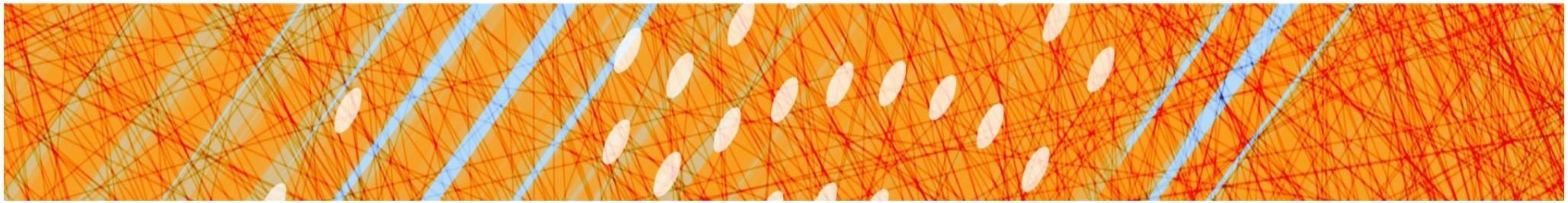


Lectures 16 & 17:
Classes in Action!
(Chapter 17)

CS 1110

Introduction to Computing Using Python

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]

How is A4 going?

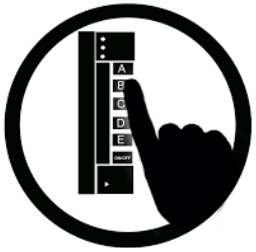
(A) I haven't started.

(B) I've started, but I am still working on my first function.

(C) I have 1 of the 3 functions finished!

(D) I have 2 of the 3 functions finished!

(E) I'm DONE, baby!



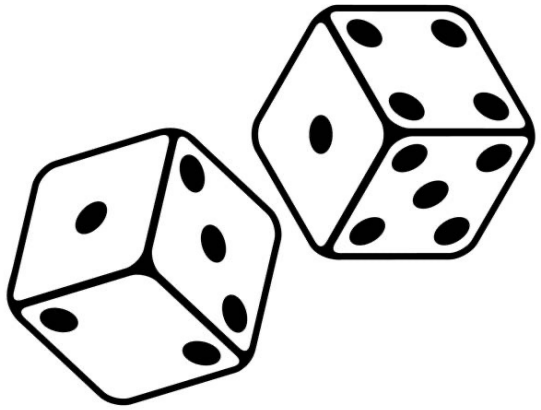
Last time we learned how to make:

- Class definitions
- Class specifications
- Class attributes
- The `__init__` function
- Instance attributes (using `self`)
- Instance methods

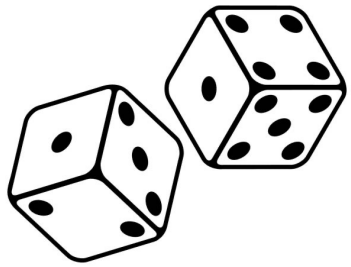
Designing Types

- **Type**: set of values and the operations on them
  - int: (**set**: integers; **ops**: +, –, *, /, …)
  - Point2 (**set**: x,y coordinates; **ops**: distanceTo, …)
  - Card (**set**: suit * rank combinations; **ops**: ==, !=, <)
  - Others to think about: Person, Student, Image, Date, *etc.*

- To define a class, think of a ***type*** you want to make



Making a Class into a Type



1. What values do you want in the set?
   - What are the attributes? What values can they have?
   - Are these attributes shared between instances (class attributes) or different for each instance (instance attributes)?
   - What are the *class invariants:* things you promise to keep true **after every method call**
2. What operations do you want?
   - This often influences the previous question
   - What are the *method specifications:* states what the method does & what it expects (preconditions)
   - Are there any special methods that you will need to provide?

Write your code to make it so!

Implementing a Class

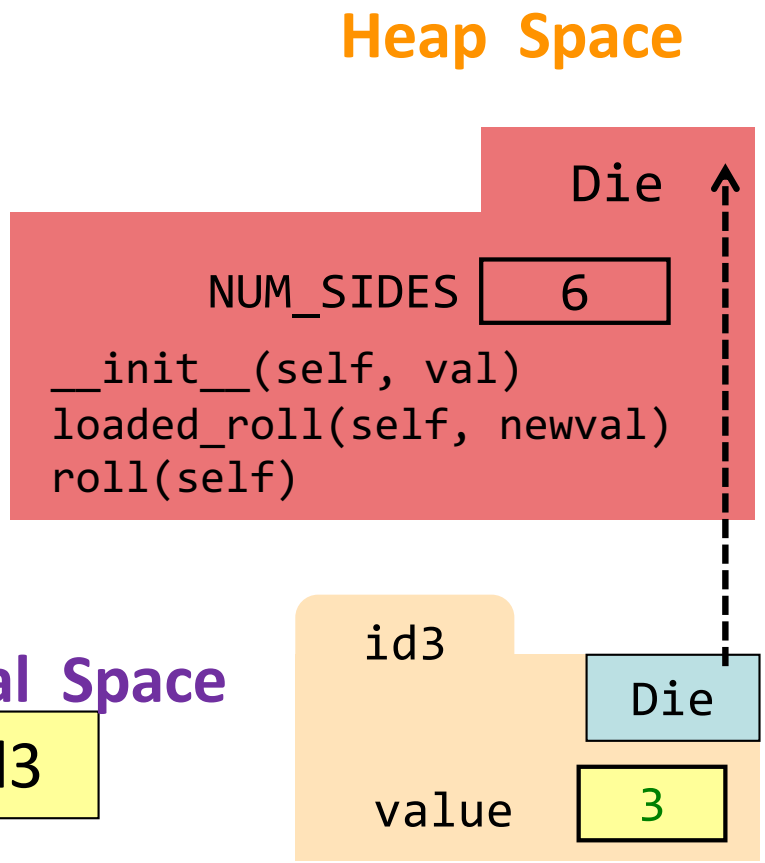
- After deciding on class & instance attributes, all that remains is to fill in the methods. (All?!)
- When ***implementing*** methods:
  1. Assume preconditions are true (*checking is friendly*)
  2. Assume class invariant is true to start
  3. Ensure method specification is fulfilled
  4. Ensure class invariant is true when done
- Later, when ***using*** the class:
  - When calling methods, ensure preconditions are true
  - If attributes are altered, ensure class invariant is true

Go Implement the Die Class…

- Class definitions
- Class specifications
- Class attributes
- The `__init__` function
- Instance attributes (using `self`)
- Instance methods

Name Resolution for Objects (attributes)

- `myobject.myattribute` means
  - Go the folder for `myobject`
  - Find method `myattribute`
  - If missing, check class folder
  - If not in either, raise error



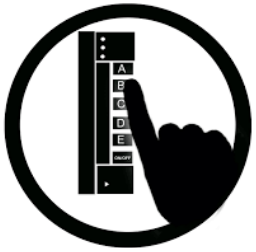

```
d1 = Die(3)
# finds attribute in object folder
print(d1.value)
# finds attribute in class folder
print(d1.NUM_SIDES)
```

← works, but dangerous… why?

What gets Printed? (Q)

```
import dice

d1 = dice.Die()
d2 = dice.Die()
print(d1.NUM_SIDES)
print(d2.NUM_SIDES)
print(dice.Die.NUM_SIDES)

d1.NUM_SIDES = 12
print(d1.NUM_SIDES)
print(d2.NUM_SIDES)
print(dice.Die.NUM_SIDES)
```

A:	B:	C:	D:
6	6	6	6
6	6	6	6
6	6	6	6
12	12	12	6
12	12	6	6
12	6	6	6



Accessing vs. *Modifying* Class Variables

- **Recall:** you cannot assign to a global variable from inside a function call
- **Similarly:** you cannot assign to a class attribute from "inside" an object variable

```
d1 = Die()
Die.NUM_SIDES = 12        # updates class attribute
d1.NUM_SIDES = 24         # creates new object attribute
                          #    called NUM_SIDES
```

Better to refer to Class Variables

using the Class Name

Just like it did in the __init__ method!

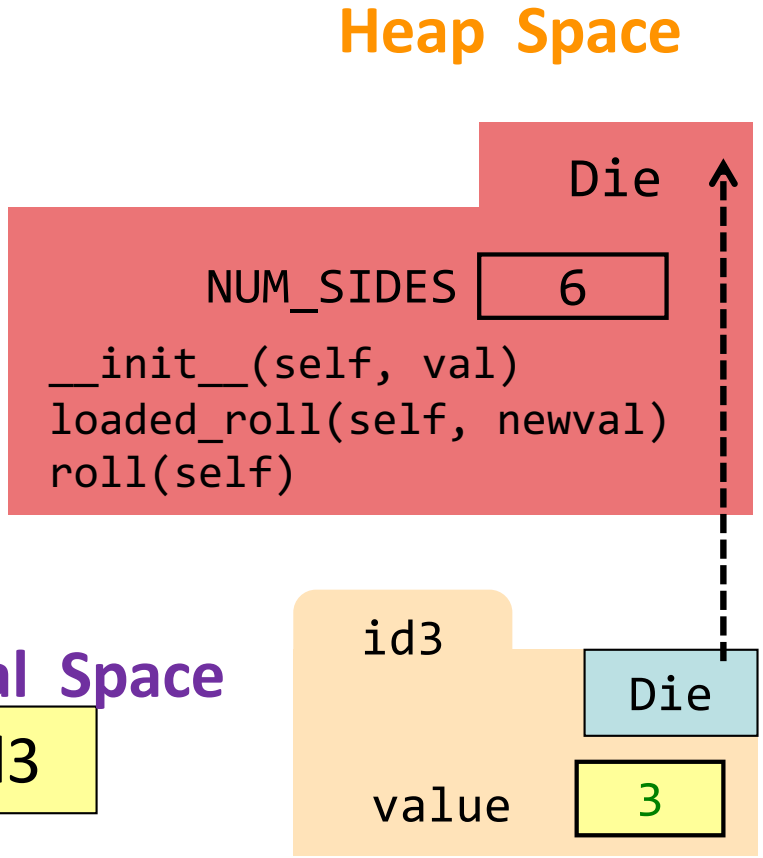
Lesson #1

1. Refer to Class Attributes using the Class Name

```
d1 = Die()
print("this die has " + str(Die.NUM_SIDES) + " sides")
```

Name Resolution for Objects (methods)

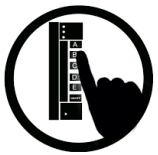
- `myobject.mymethod()` means
  - Go the folder for `myobject`
  - Find method `mymethod`
  - If missing, check class folder
  - If not in either, raise error




```
d1 = Die(3)
d1.roll()
```

If method lives in the class folder, why not call the methods like this:

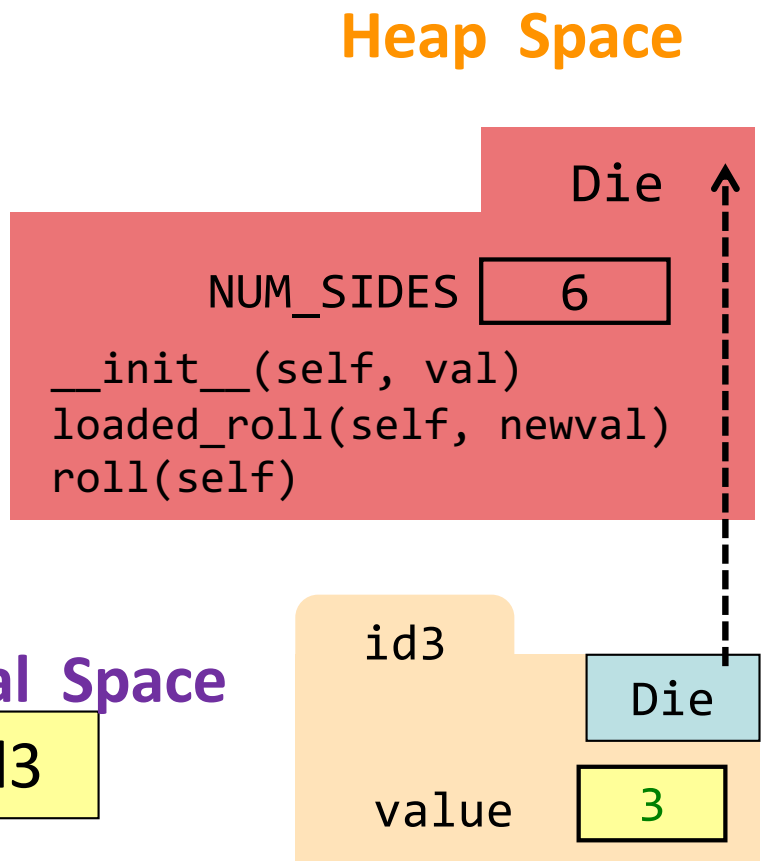
`Die.roll()` ?



(A) We should call it Die.roll()
(B) Calling Die.roll() works but is bad style
(C) Calling Die.roll() wont' work
(D) I don't know

Name Resolution for Objects (methods)

- `myobject.mymethod()` means
  - Go the folder for `myobject`
  - Find method `mymethod`
  - If missing, check class folder
  - If not in either, raise error




```
d1 = Die(3)
d1.roll()
```

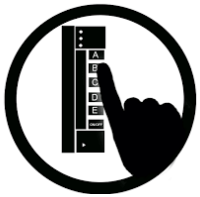
If method lives in the class folder, why not call the methods like this:

`Die.roll()` ?

Because most methods operate on instance attributes & need `self`.

Lesson #2

2. Don’t forget `self`
   - in parameter list of method (method header)
   - when defining method (method body)



If you forget self, Error #1

```
# you forget self entirely
def loaded_roll(self, newval):
  self.value = newval
```

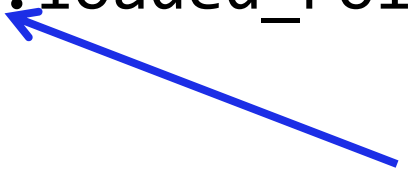
```
d1 = Die()
d1.loaded_roll(5)
```

(A)TypeError: loaded_roll() takes 1 positional argument but 2 were given
(B)NameError: name ‘value' is not defined
(C) There is no error!
(D) I don't know.

If you forget self, Error #1

```
# you forget self entirely
def loaded_roll(self, newval):
  self.value = newval
```

```
d1 = Die()
d1.loaded_roll(5)
```



always passes d1 as first argument!

```
TypeError: loaded_roll() takes 1 positional
argument but 2 were given
```

If you forget self, Error #2 (reading)

```
# you forget self in the body
def loaded_roll(self, newval):
  print("old value was " + str(self.value))
  self.value = newval
```

```
d1 = Die()
d1.loaded_roll(5)
```

```
NameError: name 'value' is not defined
```

If you forget `self`, Error #3 (writing)

```
# you forget self in the body
def loaded_roll(self, newval):
      self.value = newval
```

```
d1 = Die()
d1.loaded_roll(5)
```

Worst kind of error: No ERROR.
(code just silently doesn't work…)

`__init__` is just one of many Special Methods

Start/end with 2 underscores

- This is standard in Python
- Used in all special methods
- Also for special attributes

`__init__` for initializer

`__str__` for `str()`

`__eq__` for ==

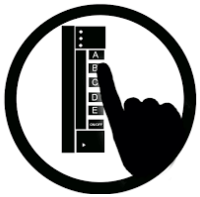
`__lt__` for <, …

```
class Die():
    """Instances are 6-sided dice."""

    def __init__(self):
        <snip>

    def __str__(self):
        """Returns: string version of die"""


    def __eq__(self, other):
        """Returns: True if ... ? """
```

See Fractions example at the end of this lecture

Optional: for a complete list, see

https://docs.python.org/3/reference/datamodel.html#basic-customization



What is equality?

```
d1 = Die(2)
d2 = Die(2)

x = (d1 == d2)
```

Are they equal?

(does x hold the value True or False?)

(A) Python will say they are equal and I think they are equal.

(B) Python will say they are equal but I don't think they are equal.

(C) Python will not say they are equal but I think they are equal.

(D) Python will not say they are equal and I don't think they are equal.

(E) Huh?

Go implement more special methods for our Die class.

The remaining slides will not be covered in lecture but walk you through another example of developing your own class.

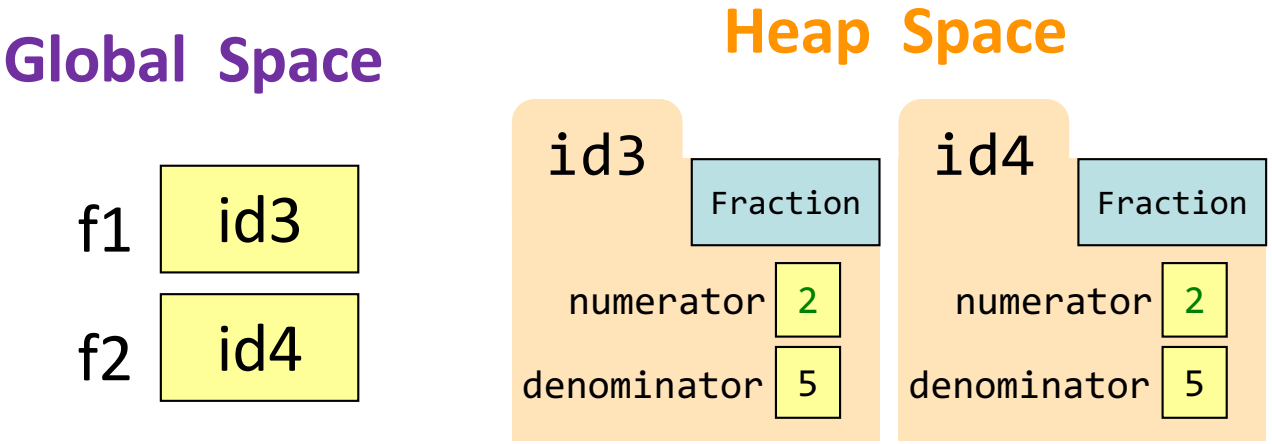
Planning out a Class: Fraction

- What *attributes*? What *invariants*?
- What *methods*? What *initializer?* other *special methods*?

```
class Fraction:
    """Instance is a fraction n/d
     Attributes:
        numerator:   top [int]
        denominator: bottom [int > 0]  """

    def __init__(self, n=0, d=1):
        """Init: makes a Fraction"""
        assert type(n)==int
        assert type(d)==int and d>0
        self.numerator = n
        self.denominator = d
```

What is equality?

```
f1 = Fraction(2,5)
f2 = Fraction(2,5)
if f1 == f2:
  # do we go here?
else:
  # or here?
```



By default, == compares ***folder IDs***

Operator Overloading: Equality

Implement `__eq__` to check for equivalence of two `Fraction`s instead

```
class Fraction():
    """Instance attributes:
        numerator:    top       [int]
        denominator: bottom [int > 0]"""

    def __eq__(self,q):
      """Returns: True if self, q equal,
        False if not, or q not a Fraction"""
        if type(q) != Fraction:
            return False
        left = self.numerator*q.denominator
        right = self.denominator*q.numerator
        return left == right
```

Problem: Doing Math is Unwieldy

What We Want

$$\left(\frac{1}{2}+\frac{1}{3}+\frac{1}{4}\right)*\frac{5}{4}$$

Why not use the standard Python math operations?

What We Get

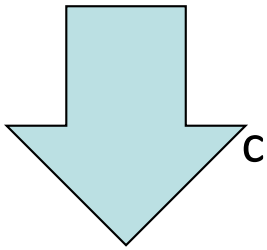
```
>>> p = Fraction(1,2)
>>> q = Fraction(1,3)
>>> r = Fraction(1,4)
>>> s = Fraction(5,4)
>>> (p.add(q.add(r))).mult(s)
```

Seriously?

Operator Overloading: Addition

```
class Fraction():
    """Instance attributes:
           numerator:    top        [int]
           denominator: bottom [int > 0]"""”

    def __add__(self,q):
        """Returns: Sum of self, q
        Makes a new Fraction
        Precondition: q a Fraction"""
        assert type(q) == Fraction
        bot = self.denominator*q.denominator
        top = (self.numerator*q.denominator+
               self.denominator*q.numerator)
        return Fraction(top,bot)
```

```
>>> p = Fraction(1,2)
>>> q = Fraction(3,4)
>>> r = p+q
```



Python converts to

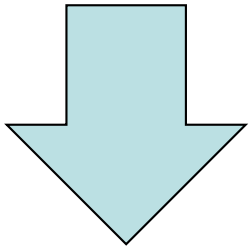
```
>>> r = p.__add__(q)
```

Operator overloading uses method in object on left.

Operator Overloading: Multiplication

```
class Fraction():
    """Instance attributes:
        numerator:    top        [int]
        denominator: bottom [int > 0]"""

    def __mul__(self,q):
        """Returns: Product of self, q
        Makes a new Fraction; does not
        modify contents of self or q
        Precondition: q a Fraction"""
        assert type(q) == Fraction
        top = self.numerator*q.numerator
        bot = self.denominator*q.denominator
        return Fraction(top,bot)
```

```
>>> p = Fraction(1,2)
>>> q = Fraction(3,4)
>>> r = p*q
```



Python converts to

```
>>> r = p.__mul__(q)
```