http://www.cs.cornell.edu/courses/cs1110/2022sp

Lecture 16:
More on Classes
(Chapter 17)

CS 1110

Introduction to Computing Using Python

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

Announcements

- Prelim 2 alternate time request form live due 4/1
  - Are you enrolled in? CHEM 2090, AEM 2601, ECON 1120, HADM 1360 → **FILL OUT THE SURVEY!**

- To reduce wait times during consulting hours:

  If wait time exceeds 20 mins, we will shift to a 15-minutes-per-student system.

- Remember to reach out to your lab leads for lab-related support. (https://www.cs.cornell.edu/courses/cs1110/2022sp/timeplace/)

We know how to make:

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

Go back to previous lecture

Go over the "Rules to live by" slides

`__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 Point2():
    """Instances are points in 2D space"""

    def __init__(self,x=0,y=0):
        <snip>

    def __str__(self):
        """Returns: string with contents"""
        return '(' + str(self.x) + ', ' +
                     str(self.y) + ')'

    def __eq__(self, other):
        """Returns: True if both coords equal"""
        return self.x == other.x
            and self.y == other.y
```

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

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** *(see* `n_credit` *invariant)*
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!

Planning out a Class: Fraction

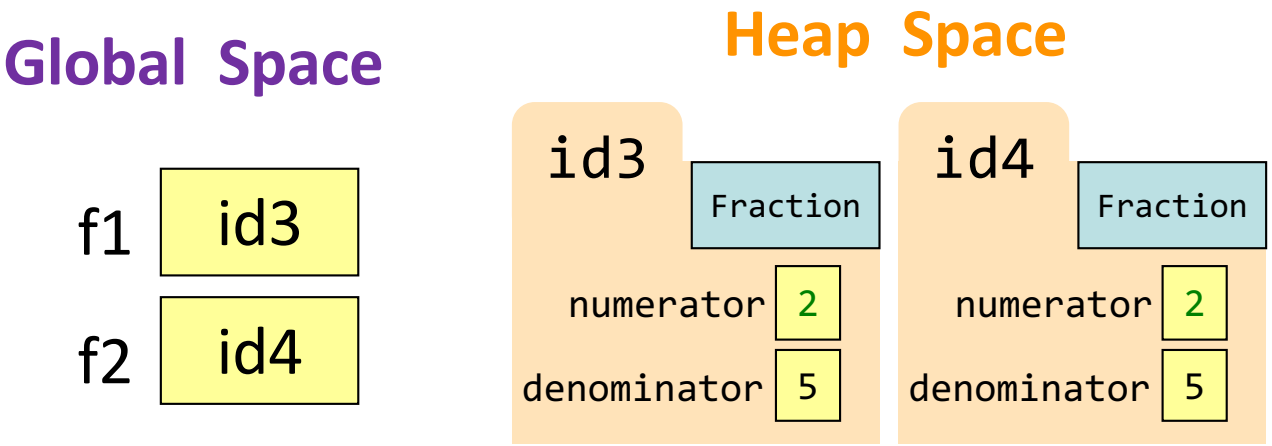
- What *attributes*? What *invariants*?
- What *methods*? What *initializer?* other *special methods*?

```
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)
```

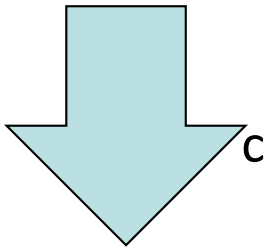
Seriously?

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)
```

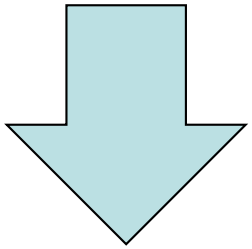
Operator overloading uses method in object on left.

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)
```