Introduction to Object-Oriented Programming (OOP) II

Quick review: operator overloading

- We have learned some basic features of OOP
  - Constructor: `def __init__(self):`
  - String representation: `def __str__(self):`, or `def __repr__(self):`
  - Method within a class: `def tomorrow(self):`
  - Object attributes (object variables …) `self.year, self.month, self.day.`
- We also discussed the topic of operator overloading

What does it mean?

- An operator such as '==', '>' can be associated with a function to reflect its meaning.
- E.g., in our Date class, we have three functions
  - is_equal(), is_before(), is_after()
  - When comparing two Date objects, we'd say d1.is_equal(d2), d1.is_before(), d1.is_after()
- If we implement operator overloads for the Date class, we could have said
  - d1 == d2, d1 < d2, d1 > d2

Overloading '=='

```
class Date:
    …
    def __eq__(self, other):
         if self.year == other.year and \
             self.month == other.month and \
             self.day == other.day:
             return True
         else:
             return False
```

If the function is_equal() has been defined, we can do …

```
class Date:
    …
    def __eq__(self, other):
         if self.is_equal(other):
            return True
         else:
            return False
```

```
class Date:
    …
    def __eq__(self, other):
         return self.is_equal(other)
```

Overloading '>'

```
class Date:
    …
    def __gt__(self, other):
         return self.is_after(other)
```

Overloading '>='

```
class Date:
    …
    def __ge__(self, other):
         return self.is_after(other) or \
              self.is_equal(other)
```

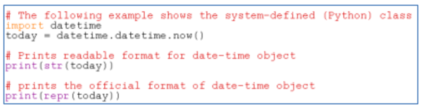
__str__() vs __repr__()

- In Python, when printing an object, two methods can play roles, __str__() and __repr__()
- The difference is illustrated well by this post, though the syntax of the post is Python 2.x

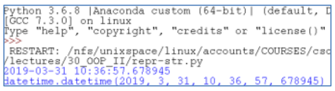
https://www.geeksforgeeks.org/str-vs-repr-in-python

- Let us walk through the example

The **datetime** class is provided by Python, in which the __repr__() and __str__() are already defined.

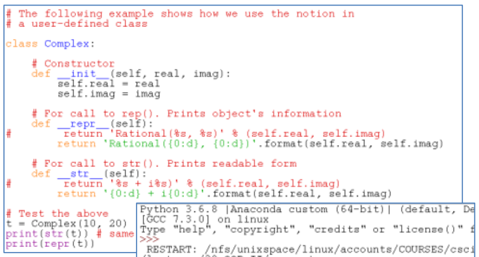


```
# The following example shows the system-defined (Python) class
import datetime
today = datetime.datetime.now()

# Prints readable format for date-time object
print(str(today))

# prints the official format of date-time object
print(repr(today))
```

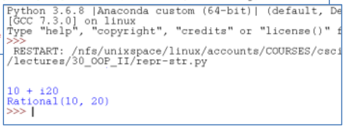


```
Python 3.6.8 |Anaconda custom (64-bit)| (default, D
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license()"
>>>
 RESTART: /nfs/unixspace/linux/accounts/COURSES/csc
/lectures/30_OOP_II/repr-str.py
2019-03-31 10:36:57.678945
datetime.datetime(2019, 3, 31, 10, 36, 57, 678945)
```

The **Complex** class is defined by the programmer (YOU!), in which the __repr__() and __str__() are defined at your wish.



```
# The following example shows how we use the notion in
# a user-defined class

class Complex:

    # Constructor
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    # For call to rep(). Prints object's information
    def __repr__(self):
#        return 'Rational(%s, %s)' % (self.real, self.imag)
        return 'Rational({0:d}, {0:d})'.format(self.real, self.imag)

    # For call to str(). Prints readable form
    def __str__(self):
#        return '%s + i%s)' % (self.real, self.imag)
        return '{0:d} + i{0:d}'.format(self.real, self.imag)

# Test the above
t = Complex(10, 20)
print(str(t)) # same
print(repr(t))
```



```
Python 3.6.8 |Anaconda custom (64-bit)| (default, De
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license()" f
>>>
 RESTART: /nfs/unixspace/linux/accounts/COURSES/csci
/lectures/30_OOP_II/repr-str.py


10 + i20
Rational(10, 20)
>>> |
```

Show repr-str.py

Class exercises

Given a Book class as follows, define methods to overload '>', '<', '>=', '<=', and '==', if the comparison is based on the attribute 'pub_year'

```
class Book:

    def __init__( self, title, author, pub_year ):
        '''
        Create an object
        '''
        self.author = author
        self.title  = title
        self.pub_year = pub_year    # an integer
```

Class exercises

If the comparison is based on the attribute 'title', write the method that overloads '>' using string comparison.

```
class Book:

    def __init__( self, title, author, pub_year ):
        '''
        Create an object
        '''
        self.author = author     # a string
        self.title  = title      # a string
        self.pub_year = pub_year  # an integer
```

Class exercises

If the comparison is based on the attribute 'pub_year', if 'pub_year' is the same, then check 'title', if title is the same, check 'author'.

```
class Book:

    def __init__( self, title, author, pub_year ):
        '''
        Create an object
        '''
        self.author = author     # a string
        self.title  = title      # a string
        self.pub_year = pub_year # an integer
```

Other operator overload

- Python supports more operator overload
  - **__ne__** : not equal
  - **__contains__** : membership check
  - **__add__** : add to the collection (+)
  - **__iadd_** : for +=
- See book_shelf demonstration

book.py, book_shelf.py, book_shelf_app.py