__str__

- If implemented, creates a string representation of an instance of the class used to ‘pretty print’ the instance

- Invoked explicitly with `str(object_name)`

- Invoked implicitly by functions related to printing: `print(object_name)`
Example

class Account(object):
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

    def __str__(self):
        s = 'Account holder: ' + self.name + '
        s += 'Balance: ' + str(self.balance) + '
        return s

checking = Account('David', 100000)
print(checking)

__repr__

* If implemented, creates a simple string representation of an instance that can be evaluated to recreate the instance

* Invoked explicitly with repr(object_name)

* If not implemented, a string of this form is returned:
  <account.Account instance at 0x[hex address]>
Example

```python
class Account(object):
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

    def __repr__(self):
        return 'Account({}, {})'.format(self.name, self.balance)

checking = Account('David', 100000)
repr(checking)
new_checking = eval(repr(checking))
```

Comparisons

- Python supports a number of methods that, if implemented, compare instances of a class.
- `__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__`, `__ne__`
- As the implementer, you get to decide what each of these mean with respect to instances of your class.
Example

```python
class Account(object):
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

    def __lt__(self, other):
        if self.balance < other.balance:
            return True
        elif self.balance == other.balance and self.name.lower() < other.name.lower():
            return True
        return False

Example

checking1 = Account('Alice', 10000)
checking2 = Account('Bob', 5000)
if checking1 < checking2:
    print('checking1')
else:
    print('checking2')
```

what is printed?
Example

```python
checking1 = Account('Alice', 10000)
checking2 = Account('Bob', 5000)

if checking1 < checking2:
    print(checking1)
else:
    print(checking2)
```

what is printed?

In this version, there are no " in the print statements. So `__str__` is invoked (if implemented) and used to print the appropriate Account object.

Example

```python
class Account(object):
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

    def __eq__(self, other):
        if self.balance == other.balance and self.name == other.name:
            return True
        return False
```
Math Operations

- Python supports a number of methods that, if implemented, perform math operations on instances of a class.
- `__add__`, `__sub__`, `__mul__`, `__div__`, `__mod__`, `__iadd__`, `__isub__`, `__abs__`, `__int__` (and more)
- Some (or all) of these won’t make sense for some classes.

Math Operations

- `__add__`, `__sub__`, `__mul__`, `__div__`, `__mod__`, etc override `+`, `-`, `*`, `/`, `%`
  - They must return the result of the operation
- `__iadd__`, `__isub__`, `__imul__`, `__idiv__`, `__imod__`, etc override `+=`, `-=`, `*=` , `/=`, `%=`
  - They must return `self` when complete
Example

```python
class Account(object):
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

    def __add__(self, other):
        return self.balance + other.balance

    def __mul__(self, other):
        return self.balance * other.balance
```

Example

```python
class Account(object):
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

    def __iadd__(self, other):
        self.balance += other.balance
        return self

    def __imul__(self, other):
        self.balance *= other.balance
        return self
```
Callable Interface

- Implementing `__call__` allows an object to be used like a function. This is also known as a functor. One advantage is that a functor contains state (stored in the attributes). This state can be used to affect what the functor does.

- Why? It can be useful but really, it’s just cool.

Example

```python
class Account(object):
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

    def __call__(self, n):
        self.balance += n
```
Another Example

class Multiplier(object):
    def __init__(self, factor):
        self.factor = factor

    def __call__(self, n):
        return n * factor

Rate3 = Multiplier(3)
Rate5 = Multiplier(5)

fives = map(Rate5, [randint(1, 20) for _ in range(10)])

__del__

- __del__ is called automatically (if it exists) when an object is being destroyed
- Only implement this if there is some resource cleanup needed:
  - release locks
  - close a connection
  - update class instance variables
__bool__

- __bool__ is called automatically when a truth-value test is performed on an object
- if it is not implemented, __len__ is used instead

Example

```python
class Account(object):
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

    def __bool__(self, n):
        return len(self.name) > 0
```
Example

class Account(object):
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

    def __bool__(self, n):
        return self.balance >= 0