Class #24: Special Methods

__str__

- If implemented, creates a string representation of an instance of the class
- Invoked explicitly with \texttt{str(object\_name)}
- Invoked implicitly by functions related to printing: \texttt{print(object\_name)}

Example

```python
class Account(object):
    def \_init\_(self, name, balance):
        self.name = name
        self.balance = balance
    def \_str\_(self):
        s = 'Account holder: ' + self.name + '\n'
        s += 'Balance: ' + \texttt{str(self.balance)} + '\n'
        return s

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

Example
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 # why?
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 __mul__(self, other):
        return self.balance * other.balance  # why?!
```

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.

Another Example

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

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

Rate3 = Multiplier(3)
Rate5 = Multiplier(5)
fives = map(Rate5, [randint(1, 20) for _ in range(10)])```