Dunder Methods

Thomas Schwarz, SJ

Dunder Methods

- Python reserves special names for functions that allows the programmer to emulate the behavior of built-in types
    - For example, we can create number like objects that allow for operations such as addition and multiplication
        - These methods have special names that start out with two underscores and end with two underscores

- Aside: If you preface a variable / function / class with a single underscore, you indicate that it should be treated as reserved and not used outside of the module / class

Dunder Method

- A class for playing cards:
  - A card has a suit and a rank
    - We define this in the constructor __init__

```
class Card:
    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank
```

Dunder Method

- We want to print it
  - Python likes to have two methods:
    - __repr__ for more information, e.g. errors
    - __str__ for the print-function
    - Both return a string

```
class Card:

    def __str__(self):
        return self.suit[0:2]+self.rank[0:2]
    def __repr__(self):
        return "{}-{}".format(self.suit, self.rank
```

Dunder Method

- __repr__ is used when we create an object in the terminal

```
>>> Card("Heart", "Queen")
Heart-Queen
```

- __str__ is used within print or when we say `str(card)`

```
>>> print(Card("Heart", "Queen"))
HeQu
>>> str(Card("Heart", "Queen"))
'HeQu'
```

Dunder Method

- We now create a carddeck class
  - Consists of a set of cards
  - Constructor uses a list of ranks and a list of suits

```
class Deck:
    def __init__(self, los, lov):
        self.cards = [Card(suit, rank) for suit in los
                                       for rank in lov]
```

Dunder Method

- We create the string method. Remember that it needs to return a string.

```
class Deck:
    def __init__(self, los, lov):
        self.cards = [Card(suit, rank) for suit in los
                                       for rank in lov]
    def __str__(self):
        result = []
        for card in self.cards:
            result.append(str(card))
        return " ".join(result)
```

Dunder Method

- In order to allow python to check whether a deck exists, we want to have a length class. Besides, it is useful in itself.
  - `if deck:` works by checking `len(deck)`

```
class Deck:

    def __len__(self):
        return len(self.cards)
```

Dunder Method

- Given a deck, we want to be able to access the i-th element.
- We do so by defining __getitem__

```
class Deck:

    def __getitem__(self, position):
        return self.cards[position]
```

Dunder Method

- This turns out to be very powerful:

```
french_deck = Deck(['Spade', 'Diamonds', 'Hearts', 'Clubs'],
                   ['Ace', 'King', 'Queen', 'Jack', '10', '9',
                    '8', '7', '6', '5', '4', '3', '2'])
```

- We can print out the i-th element of the deck

```
>>> str(french_deck[5])
'Sp9'
```

- But we can also **slice** the deck

```
>>> print(french_deck[6:12])
[Spade-8, Spade-7, Spade-6, Spade-5, Spade-4, Spade-3]
```

Dunder Method

- We can use random.choice() to select a card

```
>>> random.choice(french_deck)
Diamonds-9
```

- Only for random.sample do we need to go to the underlying instance field

```
>>> random.sample(french_deck.cards, 5)
[Hearts-8, Hearts-2, Hearts-Ace, Hearts-6, Diamonds-Ace]
>>> random.sample(french_deck.cards, 5)
[Hearts-5, Clubs-Queen, Diamonds-Ace, Clubs-3, Clubs-King]
```

  - But this is ugly and we better write a class method for it.