Lecture 8-1

A Few More OOP methods and Some Pythonic Features

Week 8 Monday

Miles Chen, PhD

The `Card` class so far

The `Card` class has an `__init__` method which assigns a numeric value to the suit and to the rank.

It has a few methods:

- `__str__` which is used to show the card in a user-friendly form.
- `__lt__` which is used for comparison and allows card objects to be sorted
- `__eq__` which is used to test equality

In [1]:

```
class Card:
    def __init__(self, suit = 0, rank = 2):
        self.suit = suit
        self.rank = rank

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King']

    def __str__(self):
        return "%s of %s" % (Card.rank_names[self.rank],
                             Card.suit_names[self.suit])

    def __lt__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 < t2

    def __eq__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 == t2
```

The Deck class so far

The `Deck` class has a few methods:

- `__init__` method which creates 52 cards and stores them in a list `self.cards`
- `__str__` which iterates through all items in the `self.cards` list and prints them
- `pop_card` which pops the last card in the list `self.cards`
- `add_card` which appends a card in the list `self.cards`
- `shuffle` which shuffles the list `self.cards`
- `sort` which sorts the list `self.cards`. It is able to do this because the `Card` objects have `__lt__` which allows for comparison
- `move_cards` which moves cards from the deck to a hand.

Note: `shuffle` requires us to import the `random` module into Python

In [2]:
```
import random
random.seed(10)
```

In [3]:

```
class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1,14):
                card = Card(suit, rank)
                self.cards.append(card)
    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)

    def pop_card(self):
        return self.cards.pop()

    def add_card(self, card):
        self.cards.append(card)

    def shuffle(self):
        random.shuffle(self.cards)

    def sort(self):
        self.cards.sort()

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

The Hand class so far

The `Hand` class inherits from the `Deck` class, so it learns all of the same methods.

We change the `__init__` method so the hand starts off empty. We also provide the hand a label.

The Hand class so far

The `Hand` class inherits from the `Deck` class, so it learns all of the same methods.

We change the `__init__` method so the hand starts off empty. We also provide the hand a label.

In [4]:

```
class Hand(Deck):
    def __init__(self, label = ""):
        self.cards = []
        self.label = label
```

In [5]:

```
deck = Deck()
hand = Hand('new hand')
```

In [5]:

```
deck = Deck()
hand = Hand('new hand')
```

In [6]:

```
deck.move_cards(hand, 5)
```

In [5]:

```
deck = Deck()
hand = Hand('new hand')
```

In [6]:

```
deck.move_cards(hand, 5)
```

In [7]:

```
print(hand)
```

```
King of Spades
Queen of Spades
Jack of Spades
10 of Spades
9 of Spades
```

Current limitation

Even though we have a string representation of the card, when we create a card, the object itself is represented as an object in memory.

In [8]:
```
card1 = Card()
card2 = Card(3, 11)
```

In [9]:
```
card1
```

Out[9]:
```
<__main__.Card at 0x1835128bd88>
```

In [10]:
```
print(card1)
```

```
2 of Clubs
```

In [11]:
```
card2
```

Out[11]:
```
<__main__.Card at 0x1835128bf88>
```

In [12]:
```
print(card2)
```

```
Jack of Spades
```

This is even worse when looking at a deck or hand object

In [13]:

```
hand = Hand('new hand')
deck.move_cards(hand, 5)
hand.cards
```

Out[13]:

```
[<__main__.Card at 0x18351293f48>,
 <__main__.Card at 0x18351293f08>,
 <__main__.Card at 0x18351293ec8>,
 <__main__.Card at 0x18351293e88>,
 <__main__.Card at 0x18351293e48>]
```

As it stands, this is completely unintelligible

The `__repr__` method

The dunder (double-underscore) method `__repr__` is used to show the 'official' representation of the card object. The output should be the command that is able to create this card object.

When we created the Jack of Spades and set it equal, we called

```
card2 = Card(3, 11)
```

Thus, `Card(3, 11)` would be the official representation of this object.

In [14]:

```
class Card:
    def __init__(self, suit = 0, rank = 2):
        self.suit = suit
        self.rank = rank

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King']

    def __str__(self):
        return "%s of %s" % (Card.rank_names[self.rank],
                             Card.suit_names[self.suit])

    def __repr__(self):
        return "Card(" + str(self.suit) + ", " + str(self.rank) + ")"

    def __lt__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 < t2

    def __eq__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 == t2
```

In [15]:
```
# card2 was created under the old Card definition and does not have the __repr__ method
card2
```

Out[15]:
```
<__main__.Card at 0x1835128bf88>
```

In [16]:
```
print(card2)
```

```
Jack of Spades
```

In [17]:
```
card3 = Card(3, 11) # We create a nother jack of Spades using the new Card class with the repr metho
```

In [18]:
```
card3
```

Out[18]:
```
Card(3, 11)
```

In [19]:
```
print(card3)
```

```
Jack of Spades
```

In [20]:
```
card3 == card2
```

Out[20]:
```
True
```

In [21]:

```
# must redefine the Deck class to use the new definition of the Card class
class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1,14):
                card = Card(suit, rank)
                self.cards.append(card)
    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)

    def pop_card(self):
        return self.cards.pop()

    def add_card(self, card):
        self.cards.append(card)

    def shuffle(self):
        random.shuffle(self.cards)

    def sort(self):
        self.cards.sort()

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

In [22]:

```
class Hand(Deck):
    def __init__(self, label = ""):
        self.cards = []
        self.label = label
```

In [22]:

```
class Hand(Deck):
    def __init__(self, label = ""):
        self.cards = []
        self.label = label
```

In [23]:

```
deck = Deck()
hand = Hand('new hand')
deck.move_cards(hand, 5)
```

In [22]:

```
class Hand(Deck):
    def __init__(self, label = ""):
        self.cards = []
        self.label = label
```

In [23]:

```
deck = Deck()
hand = Hand('new hand')
deck.move_cards(hand, 5)
```

In [24]:

```
print(hand)
```

```
King of Spades
Queen of Spades
Jack of Spades
10 of Spades
9 of Spades
```

In [25]:

```
hand.cards # although not as easy to read as the string representation, the represenation makes more
```

Out[25]:

```
[Card(3, 13), Card(3, 12), Card(3, 11), Card(3, 10), Card(3, 9)]
```

How many cards are in the deck or hand?

Right now, if we want to know how many cards are in the deck or hand, we have to access the list of cards in the hand or deck directly.

In [26]:
```
hand
```
Out[26]:
```
<__main__.Hand at 0x183512b3108>
```
In [27]:
```
len(hand)
```
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-b0d5bef14cf4> in <module>
----> 1 len(hand)

TypeError: object of type 'Hand' has no len()
```
In [28]:
```
vars(hand)
```
Out[28]:
```
{'cards': [Card(3, 13), Card(3, 12), Card(3, 11), Card(3, 10), Card(3, 9)],
 'label': 'new hand'}
```
In [29]:
```
len(hand.cards)
```
Out[29]:
```
5
```

Defining the length of a class

We can fix this issue by defining the `__len__` special method, which will return the length of `self.cards`

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

In [30]:

```
class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1,14):
                card = Card(suit, rank)
                self.cards.append(card)
    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)

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

    def pop_card(self):
        return self.cards.pop()

    def add_card(self, card):
        self.cards.append(card)

    def shuffle(self):
        random.shuffle(self.cards)

    def sort(self):
        self.cards.sort()

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

In [31]:
```
class Hand(Deck):
    def __init__(self, label = ""):
        self.cards = []
        self.label = label
```

In [31]:

```
class Hand(Deck):
    def __init__(self, label = ""):
        self.cards = []
        self.label = label
```

In [32]:

```
deck = Deck()
len(deck)
```

Out[32]:

```
52
```

In [31]:

```
class Hand(Deck):
    def __init__(self, label = ""):
        self.cards = []
        self.label = label
```

In [32]:

```
deck = Deck()
len(deck)
```

Out[32]:

```
52
```

In [33]:

```
hand = Hand('new hand')
deck.move_cards(hand, 5)
```

In [31]:

```
class Hand(Deck):
    def __init__(self, label = ""):
        self.cards = []
        self.label = label
```

In [32]:

```
deck = Deck()
len(deck)
```

Out[32]: 52

In [33]:

```
hand = Hand('new hand')
deck.move_cards(hand, 5)
```

In [34]:

```
len(hand)
```

Out[34]: 5

In [31]:
```
class Hand(Deck):
    def __init__(self, label = ""):
        self.cards = []
        self.label = label
```

In [32]:
```
deck = Deck()
len(deck)
```

Out[32]:
```
52
```

In [33]:
```
hand = Hand('new hand')
deck.move_cards(hand, 5)
```

In [34]:
```
len(hand)
```

Out[34]:
```
5
```

In [35]:
```
len(deck)
```

Out[35]:
```
47
```

What if we wanted to access the first 5 cards from the deck?

In [36]:

```
deck[0:5]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-36-1d48695686e2> in <module>
----> 1 deck[0:5]

TypeError: 'Deck' object is not subscriptable
```

Right now, our deck cannot be sliced.

What if we want to iterate through the deck?

What if we want to iterate through the deck?

In [37]:

```
for card in deck:
    print(card)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-37-b03c6461f118> in <module>
----> 1 for card in deck:
      2     print(card)

TypeError: 'Deck' object is not iterable
```

Right now, our deck is not iterable.

What if we want to see the hand sorted without changing the hand?

What if we want to see the hand sorted without changing the hand?

In [38]:

```
sorted(hand)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-38-6af56c512718> in <module>
----> 1 sorted(hand)

TypeError: 'Hand' object is not iterable
```

This is not possible right now.

Making the class behave like a list or container:

https://docs.python.org/3/reference/datamodel.html#emulating-container-types

We can take the class and allow the user to slice the object as well as perform iteration.

This is achieved with the dunder method: `__getitem__(self, key)` which tells Python what to do when a particular position is requested from the class object.

In our case, we will use the `key` as an index `position`. We return the card located in the requested `[position]` from the `self.cards` list.

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

In [39]:

```
class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1,14):
                card = Card(suit, rank)
                self.cards.append(card)
    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)

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

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

    def pop_card(self):
        return self.cards.pop()

    def add_card(self, card):
        self.cards.append(card)

    def shuffle(self):
        random.shuffle(self.cards)

    def sort(self):
        self.cards.sort()

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

In [40]:

```python
deck = Deck()
print(deck)
```

```
Ace of Clubs
2 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
6 of Clubs
7 of Clubs
8 of Clubs
9 of Clubs
10 of Clubs
Jack of Clubs
Queen of Clubs
King of Clubs
Ace of Diamonds
2 of Diamonds
3 of Diamonds
4 of Diamonds
5 of Diamonds
6 of Diamonds
7 of Diamonds
8 of Diamonds
9 of Diamonds
10 of Diamonds
Jack of Diamonds
Queen of Diamonds
King of Diamonds
Ace of Hearts
2 of Hearts
3 of Hearts
4 of Hearts
5 of Hearts
```

```
6 of Hearts
7 of Hearts
8 of Hearts
9 of Hearts
10 of Hearts
Jack of Hearts
Queen of Hearts
King of Hearts
Ace of Spades
2 of Spades
3 of Spades
4 of Spades
5 of Spades
6 of Spades
7 of Spades
8 of Spades
9 of Spades
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

In [41]:

```python
# We can now perform slicing
deck[0:8]
```

Out[41]:

```
[Card(0, 1),
 Card(0, 2),
 Card(0, 3),
 Card(0, 4),
 Card(0, 5),
 Card(0, 6),
 Card(0, 7),
 Card(0, 8)]
```

In [41]:

```python
# We can now perform slicing
deck[0:8]
```

Out[41]:

```
[Card(0, 1),
 Card(0, 2),
 Card(0, 3),
 Card(0, 4),
 Card(0, 5),
 Card(0, 6),
 Card(0, 7),
 Card(0, 8)]
```

In [42]:

```python
# We can also perform iteration
for item in deck[0:8]:
    print(item)
```

```
Ace of Clubs
2 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
6 of Clubs
7 of Clubs
8 of Clubs
```

With `__getitem__` implemented, all of the slicing rules now work with our Class:

In [43]:
```
# I select the index-12th card, the King of clubs and get every 13th card after:
deck[12::13]
```

Out[43]:
```
[Card(0, 13), Card(1, 13), Card(2, 13), Card(3, 13)]
```

With `__getitem__` implemented, all of the slicing rules now work with our Class:

In [43]:
```python
# I select the index-12th card, the King of clubs and get every 13th card after:
deck[12::13]
```

Out[43]:
```
[Card(0, 13), Card(1, 13), Card(2, 13), Card(3, 13)]
```

In [44]:
```python
for item in deck[12::13]:
    print(item)
```

```
King of Clubs
King of Diamonds
King of Hearts
King of Spades
```

In [45]:

```
class Hand(Deck):
    def __init__(self, label = ""):
        self.cards = []
        self.label = label
```

In [45]:

```
class Hand(Deck):
    def __init__(self, label = ""):
        self.cards = []
        self.label = label
```

In [46]:

```
deck = Deck()
deck.shuffle()
hand = Hand('new hand')
deck.move_cards(hand, 5)
```

In [45]:
```python
class Hand(Deck):
    def __init__(self, label = ""):
        self.cards = []
        self.label = label
```

In [46]:
```python
deck = Deck()
deck.shuffle()
hand = Hand('new hand')
deck.move_cards(hand, 5)
```

In [47]:
```python
# sorted arranges by suit
for card in sorted(hand):
    print(card)
```

```
3 of Clubs
2 of Hearts
5 of Hearts
Jack of Hearts
King of Spades
```

In [45]:

```python
class Hand(Deck):
    def __init__(self, label = ""):
        self.cards = []
        self.label = label
```

In [46]:

```python
deck = Deck()
deck.shuffle()
hand = Hand('new hand')
deck.move_cards(hand, 5)
```

In [47]:

```python
# sorted arranges by suit
for card in sorted(hand):
    print(card)
```

```
3 of Clubs
2 of Hearts
5 of Hearts
Jack of Hearts
King of Spades
```

In [48]:

```python
print(hand) # original hand is left unchanged
```

```
Jack of Hearts
3 of Clubs
2 of Hearts
5 of Hearts
King of Spades
```

set item

The `__setitem__` method allows you to set items in the Class.

In our case, we can use it to assign a particular Card object to a particular position in the list of cards.

```
def __setitem__(self, key, value):
    self.cards[key] = value
```

Functions like `random.shuffle()` use the `__setitem__` method to rearrange the objects inside a container.

With `__setitem__` implemented, we can get rid of the internal `deck.shuffle()` method and simply use the `shuffle()` function.

In [49]:

```
class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1,14):
                card = Card(suit, rank)
                self.cards.append(card)
    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)

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

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

    def __setitem__(self, key, value):
        self.cards[key] = value

    def pop_card(self):
        return self.cards.pop()

    def add_card(self, card):
        self.cards.append(card)

    # no longer needed:
    # def shuffle(self):
    #     random.shuffle(self.cards)

    def sort(self):
         self.cards.sort()

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

In [50]:

```
deck = Deck()
```

In [50]:

```python
deck = Deck()
```

In [51]:

```python
for card in deck[0:10]:
    print(card)
```

```
Ace of Clubs
2 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
6 of Clubs
7 of Clubs
8 of Clubs
9 of Clubs
10 of Clubs
```

In [50]:

```
deck = Deck()
```

In [51]:

```
for card in deck[0:10]:
    print(card)
```

```
Ace of Clubs
2 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
6 of Clubs
7 of Clubs
8 of Clubs
9 of Clubs
10 of Clubs
```

In [52]:

```
random.shuffle(deck) # We can call random.shuffle() directly on deck instead of calling deck.shuffle
```

In [50]:
```
deck = Deck()
```

In [51]:
```
for card in deck[0:10]:
    print(card)
```

```
Ace of Clubs
2 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
6 of Clubs
7 of Clubs
8 of Clubs
9 of Clubs
10 of Clubs
```

In [52]:
```
random.shuffle(deck) # We can call random.shuffle() directly on deck instead of calling deck.shuffle
```

In [53]:
```
for card in deck[0:10]:
    print(card)
```

```
9 of Diamonds
5 of Spades
Ace of Clubs
King of Hearts
4 of Diamonds
10 of Spades
Jack of Diamonds
5 of Diamonds
2 of Clubs
9 of Spades
```

In [ ]:

```

```

Python Features

Taken from Chapter 19 of Think Python by Allen B Downey

Python has a number of features that are not necessary, but with them you can sometimes write code that's more concise, readable, or efficient.

Python has a number of features that are not necessary, but with them you can sometimes write code that's more concise, readable, or efficient.

Conditional Expressions

A conditional expression will check a condition and run the associated code.

The following example shows how we can ask Python to find the natural log of a number. logs do not exist for non-positive values, so if x is less than or equal to zero, we want to return `nan` instead of an error.

Python has a number of features that are not necessary, but with them you can sometimes write code that's more concise, readable, or efficient.

Conditional Expressions

A conditional expression will check a condition and run the associated code.

The following example shows how we can ask Python to find the natural log of a number. logs do not exist for non-positive values, so if x is less than or equal to zero, we want to return `nan` instead of an error.

In [54]:
```
x = -3
```

Python has a number of features that are not necessary, but with them you can sometimes write code that's more concise, readable, or efficient.

Conditional Expressions

A conditional expression will check a condition and run the associated code.

The following example shows how we can ask Python to find the natural log of a number. logs do not exist for non-positive values, so if x is less than or equal to zero, we want to return `nan` instead of an error.

In [54]:

```
x = -3
```

In [55]:

```
import math

if x > 0:
    y = math.log(x)
else:
    y = float('nan')
y
```

Out[55]:

```
nan
```

We can express the same idea more concisely with a conditional expression.

In [56]:
```
x = math.e
```

In [57]:
```
y = math.log(x) if x > 0 else float('nan')
```

In [58]:
```
y
```

Out[58]:
```
1.0
```

In [59]:
```
x = -5
y = math.log(x) if x > 0 else float('nan')
y
```

Out[59]:
```
nan
```

Recursive functions can be rewritten as conditional expressions.

In [60]:
```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

In [61]:
```
factorial(5)
```

Out[61]:
```
120
```

In [62]:
```
def factorial(n):
    return 1 if n == 0 else n * factorial(n - 1)
```

In [63]:
```
factorial(6)
```

Out[63]:
```
720
```

The conditional expression is certainly more concise. Whether it is more readable is debatable.

In general, if both branches of a conditional statement are simple expressions that are assignmened or a returned, it can be written as a conditional expression.

Variable Length Arguments and Key-Word Arguments

When we covered tuples, we saw that you can gather arguments together with `*`

In [64]:

```
def print_all(*args):
    for a in args:
        print(a)
```

In [65]:

```
print_all(1,2,3,4,5)
```

```
1
2
3
4
5
```

In [66]:

```
from random import randint

def roll(*dice):
    total = 0
    for die in dice:
        roll = randint(1, die)
        print(roll)
        total += roll
    return total
```

In [67]:

```
roll(20)
```

```
12
```

Out[67]:

```
12
```

In [68]:
```
roll(6, 6, 20)
```

```
3
4
8
```

Out[68]:
```
15
```

In [69]:
```
roll(6, 6, 20)
```

```
1
5
2
```

Out[69]:
```
8
```

In [70]:
```
roll(6, 6, 20, 20)
```

```
3
3
8
3
```

Out[70]:
```
17
```

Similarly, you can gather key-word pairs as arguments and create a function that uses them.

In [71]:
```
def print_contents(**kwargs):
    for key, value in kwargs.items():
        print ("key %s has value %s" % (key, value))
```

In [72]:
```
print_contents(CA = "California", OH = "Ohio")
```

```
key CA has value California
key OH has value Ohio
```

In [73]:

```
keys = ['CA', 'OH', 'TX', 'WA']
names = ["California", "Ohio", "Texas", "Washington"]
d = dict(zip(keys, names))
print(d)
```

```
{'CA': 'California', 'OH': 'Ohio', 'TX': 'Texas', 'WA': 'Washington'}
```

In [74]:

```
# if you want to pass a dictionary to the function, you have to use `**` to scatter them
print_contents(d)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-74-07f6714a297c> in <module>
      1 # if you want to pass a dictionary to the function, you have to use `**`
 to scatter them
----> 2 print_contents(d)

TypeError: print_contents() takes 0 positional arguments but 1 was given
```

In [75]:

```
# if you want to pass a dictionary to the function, you have to use `**` to scatter them
print_contents(**d)
```

```
key CA has value California
key OH has value Ohio
key TX has value Texas
key WA has value Washington
```

In [76]:

```
# popular use case: matplotlib
# {color = "blue", line_type = 2, line_width = 3}
# you want to make 5 plots all with the same settings
# rather than copy paste the settings into all of the plots,
# make a dictionary with the settings, and pass the dictionary using **kwargs
```

List comprehensions

List comprehensions allow us to create new lists concisely based on an existing collection

They take the form:

```
[expr for val in collection if condition]
```

This is basically equivalent to the following loop:

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

In [77]:
```python
# make a list of the squares
[x**2 for x in range(1,11)]
```
Out[77]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```
In [78]:
```python
import numpy as np
np.array([x**2 for x in range(1,11)])
```
Out[78]:
```
array([  1,   4,   9,  16,  25,  36,  49,  64,  81, 100])
```
In [79]:
```python
# square only the odd numbers
[x**2 for x in range(1,11) if x % 2 == 1]
```
Out[79]:
```
[1, 9, 25, 49, 81]
```
In [80]:
```python
# take a list of strings, and write the words that are over 2 characters long in uppercase.
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
[x.upper() for x in strings if len(x) > 2]
```
Out[80]:
```
['BAT', 'CAR', 'DOVE', 'PYTHON']
```

You can create a list comprehension from any iterable (list, tuple, string, etc)

In [81]:

```
# extract the digits from a string
string = "Hello 963257 World"
[int(x) for x in string if x.isdigit()]
# for x in string, will look at each character individually
# if x is a digit, then convert it using int()
```

Out[81]:

```
[9, 6, 3, 2, 5, 7]
```

In [82]:

```
# iterate over a dictionary's items
d = {'a':'apple', 'b':'banana', 'c':'cookie'}
```

In [83]:

```
list(d.items())  # recall what dict.items() returns: a list of tuples
```

Out[83]:

```
[('a', 'apple'), ('b', 'banana'), ('c', 'cookie')]
```

In [84]:

```
[key + ' is for ' + value for key, value in d.items() if key != 'b' ]
```

Out[84]:

```
['a is for apple', 'c is for cookie']
```

Dictionary Comprehensions

A dict comprehension looks like this:

`dict_comp = {key-expr : value-expr for value in collection if condition}`

Look at the list `strings` from above.

In [85]:

```
# create a dictionary, where the key is the word capitalized, and the value is the length of the wor
fruits = ['apple', 'mango', 'banana','cherry']
{f.capitalize():len(f) for f in fruits}
```

Out[85]:

```
{'Apple': 5, 'Mango': 5, 'Banana': 6, 'Cherry': 6}
```

In [86]:

```python
# create a dictionary where the key is the index, and the value is the string in the strings list.
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
```

In [87]:

```python
list(enumerate(strings))  # enumerate produces a collection of tuples, with index and value
```

Out[87]:

```
[(0, 'a'), (1, 'as'), (2, 'bat'), (3, 'car'), (4, 'dove'), (5, 'python')]
```

In [88]:

```python
index_map = {index:val for index, val in enumerate(strings)}
index_map
```

Out[88]:

```
{0: 'a', 1: 'as', 2: 'bat', 3: 'car', 4: 'dove', 5: 'python'}
```

In [89]:

```python
# note that enumerate returns tuples in the order (index, val)
# in the creation of a dictionary, you can swap those positions
# and even apply functions to them

# We create a dictionary where the key is the string, and the value is the index in the strings list
loc_mapping = {val : index for index, val in enumerate(strings)}
loc_mapping
```

Out[89]:

```
{'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

In [90]:

```python
index_map['a']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-90-a566f0150b5c> in <module>
----> 1 index_map['a']

KeyError: 'a'
```

In [91]:

```
loc_mapping['a']
```

Out[91]:

```
0
```

In [92]:

```
# combine dictionaries with kwargs
dd = {**loc_mapping, **index_map}
print(dd)
```

```
{'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5, 0: 'a', 1: 'as', 2:
'bat', 3: 'car', 4: 'dove', 5: 'python'}
```

In [93]:

```python
# even better... use dict.update(). This modifies the dictionary in place
loc_mapping.update(index_map)
loc_mapping
```

Out[93]:

```
{'a': 0,
 'as': 1,
 'bat': 2,
 'car': 3,
 'dove': 4,
 'python': 5,
 0: 'a',
 1: 'as',
 2: 'bat',
 3: 'car',
 4: 'dove',
 5: 'python'}
```

Generator Expressions

Generator Expressions are similar to List comprehensions.

You create them with parentheses instead of square brackets.

The result is a generator object. You can access values in the generator using `next()`

In [94]:
```
g = (n**2 for n in range(10))
```

In [95]:
```
g
```

Out[95]:
```
<generator object <genexpr> at 0x00000183513721C8>
```

In [96]:
```
next(g)
```

Out[96]:
```
0
```

In [97]:
```
next(g)
```

Out[97]:
```
1
```

In [98]:
```
next(g)
```

Out[98]:
```
4
```

In [99]:

```
for val in g:
    print(val)
```

```
9
16
25
36
49
64
81
```

In [100]:

```
next(g) # calling next after it has run out of iterations will result in an error
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-100-35efc4ce126e> in <module>
----> 1 next(g) # calling next after it has run out of iterations will result in
 an error

StopIteration: 
```

List Comprehension vs Generator Expressions in Python

A Key difference between a list comprehension and a generator is that the generator is lazy.

The list comprehension will evaluate the entire sequence of iterations. The generator will only generate the next value when it is asked to do so.

Depending on the expression that needs to be evaluated, you may prefer to use a generator over the list comprehension.

The following examples are from: https://code-maven.com/list-comprehension-vs-generator-expression

In [101]:

```
l = [n*2 for n in range(1000)] # List comprehension
g = (n*2 for n in range(1000))  # Generator expression
```

In [102]:

```
print(type(l))  # 'list'
print(type(g))  # 'generator'
```

```
<class 'list'>
<class 'generator'>
```

In [103]:

```
import sys
print(sys.getsizeof(l))  # more space in memory
print(sys.getsizeof(g))  # less space in memory
```

```
9024
120
```

In [104]:

```
# cannot access values in a generator by index
print(l[4])   # 8
print(g[4])   # TypeError: 'generator' object is not subscriptable
```

```
8
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-104-e29ce47c972b> in <module>
      1 # cannot access values in a generator by index
      2 print(l[4])   # 8
----> 3 print(g[4])   # TypeError: 'generator' object is not subscriptable

TypeError: 'generator' object is not subscriptable
```

In [105]:
```
next(g)
```
Out[105]:
```
0
```
In [106]:
```
next(g)
```
Out[106]:
```
2
```
In [107]:
```
next(g)
```
Out[107]:
```
4
```
In [108]:
```
next(g)
```
Out[108]:
```
6
```
In [109]:
```
sum(g)
```
Out[109]:
```
998988
```
In [110]:
```
sum(l)
```
Out[110]:
```
999000
```