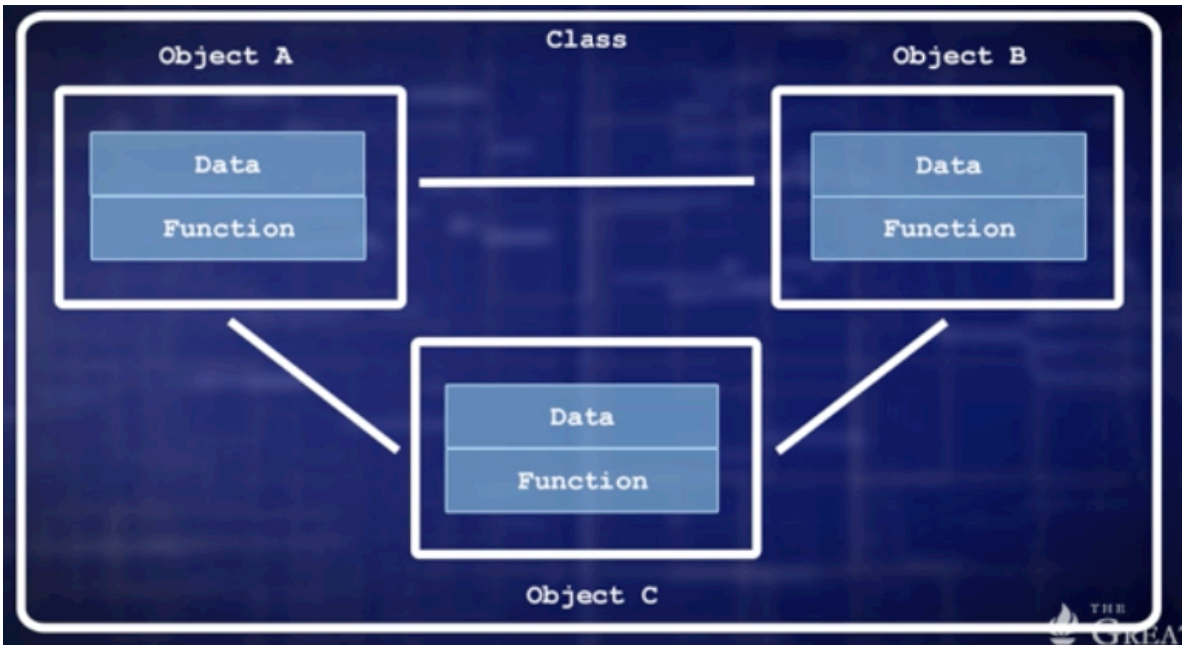
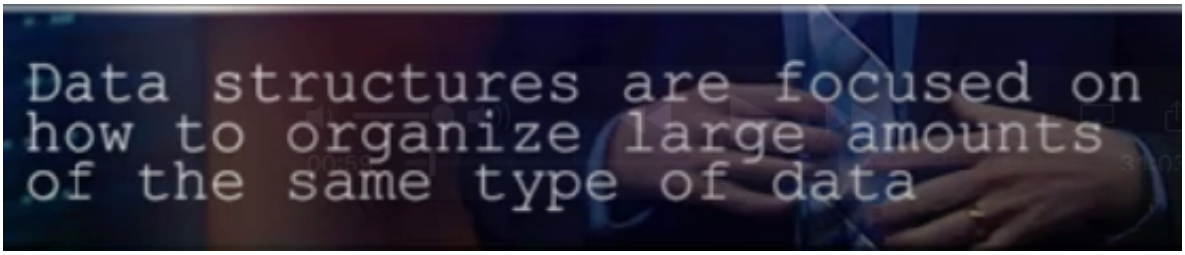
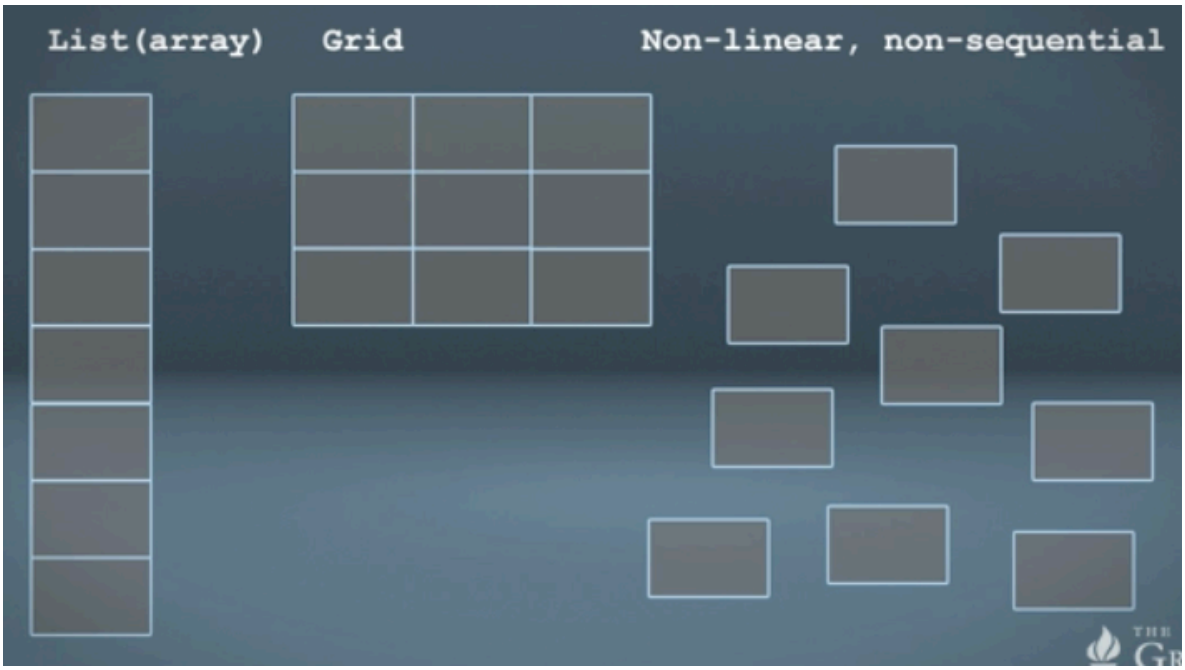
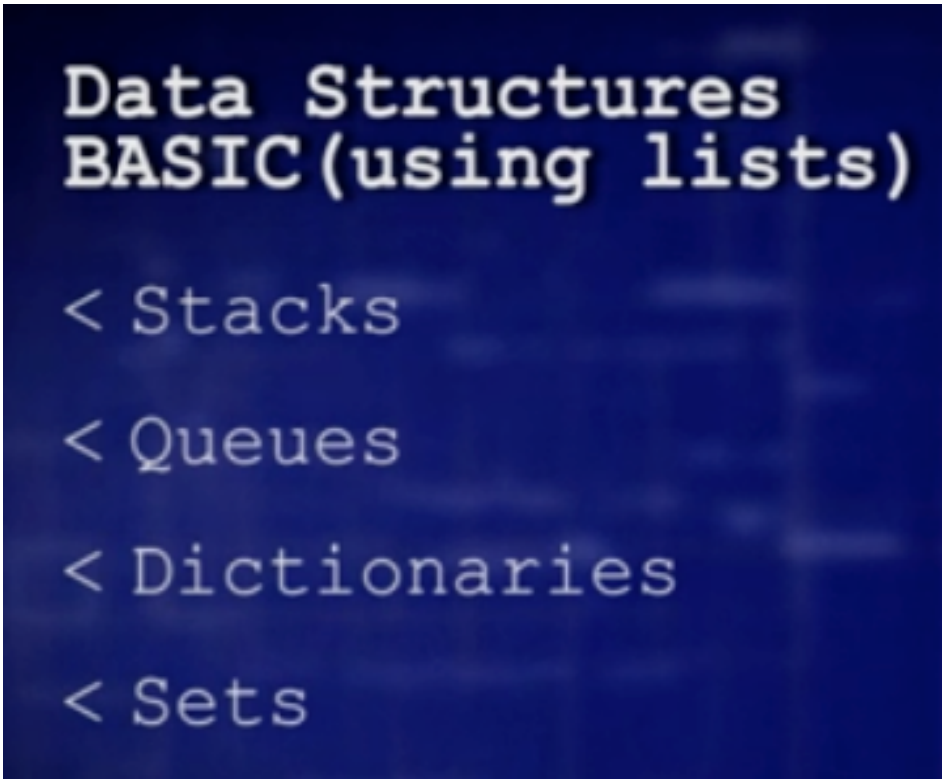
CSles19
Data Structures: Stak, Queue, Dictionary, Set

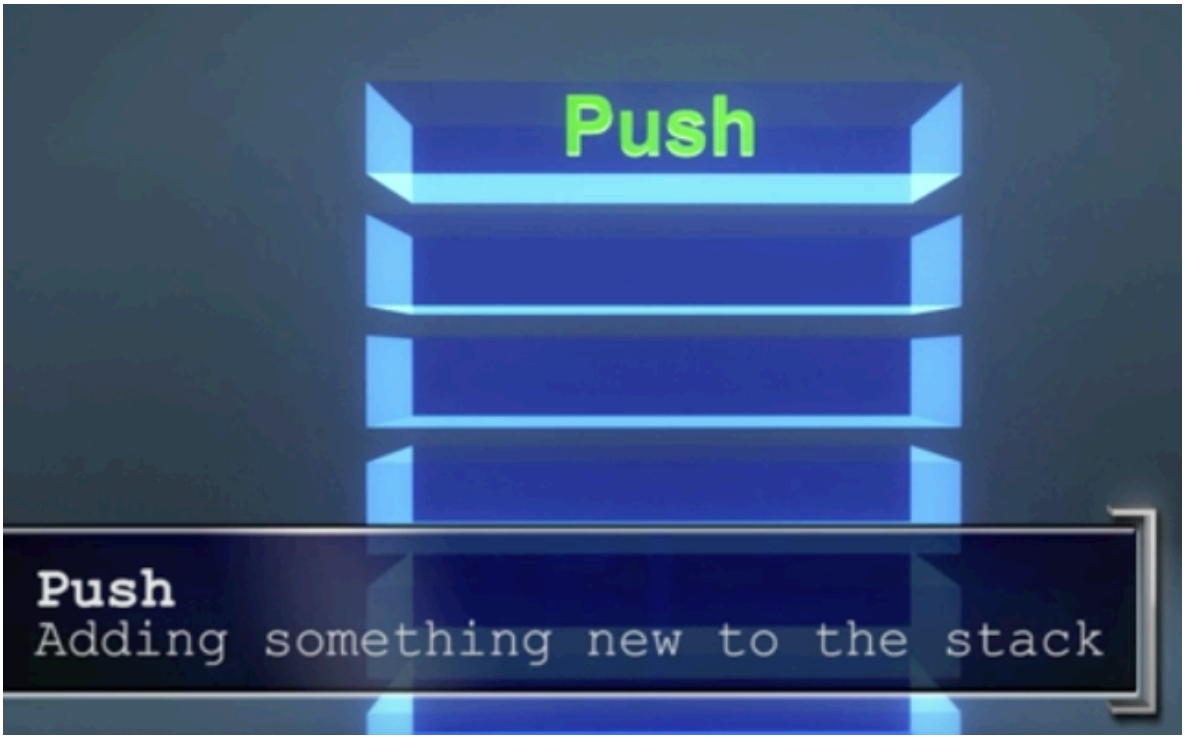


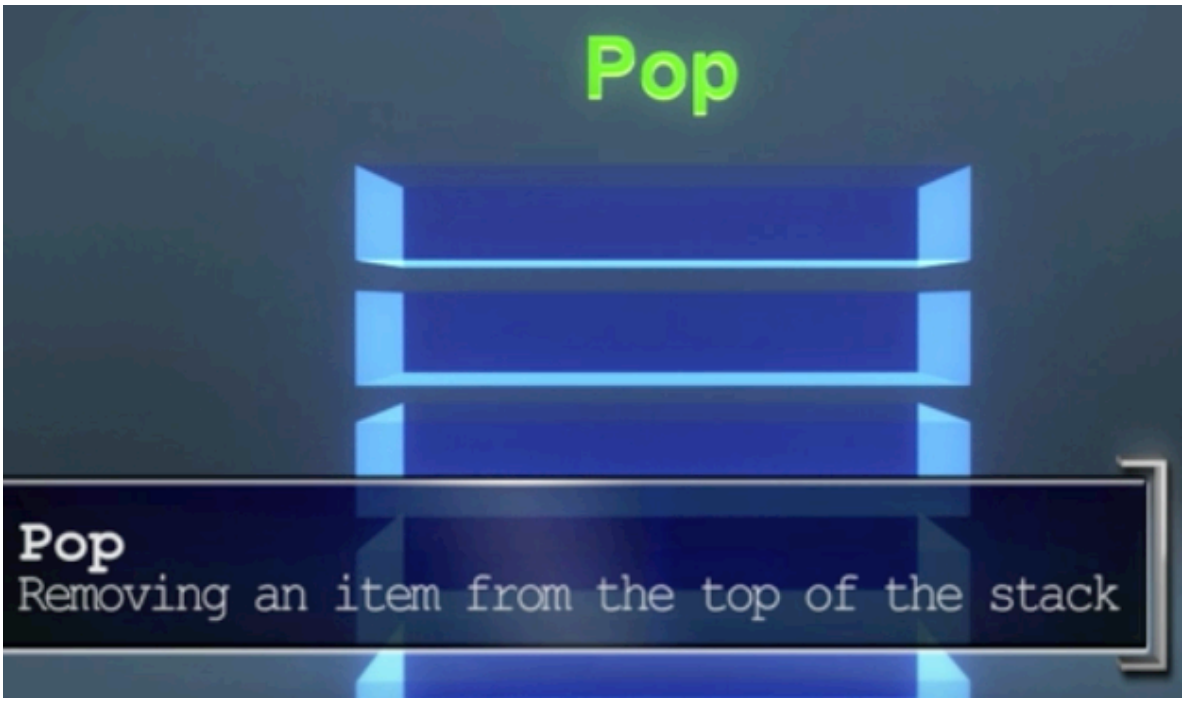





Organization affects operations


Data Structures
BASIC(using lists)
< Stacks
< Queues
< Dictionaries
< Sets

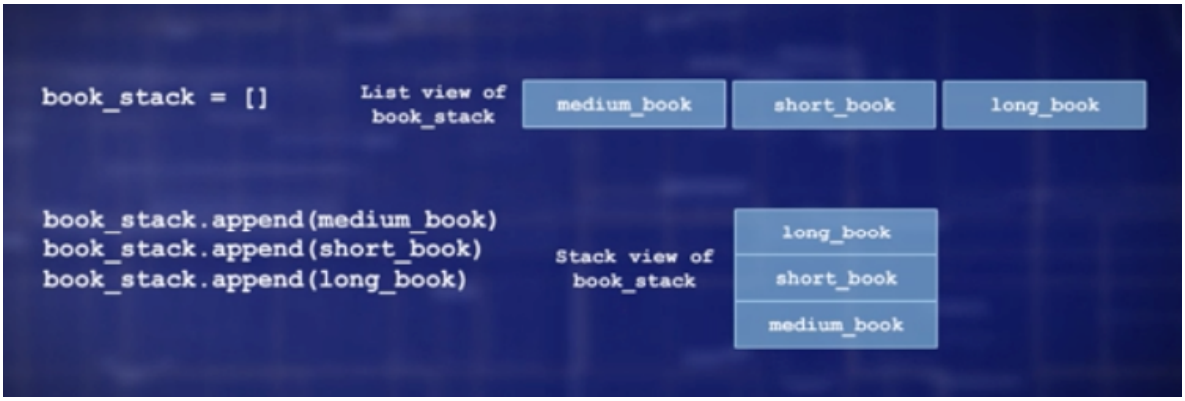

Push
Push
Adding something new to the stack

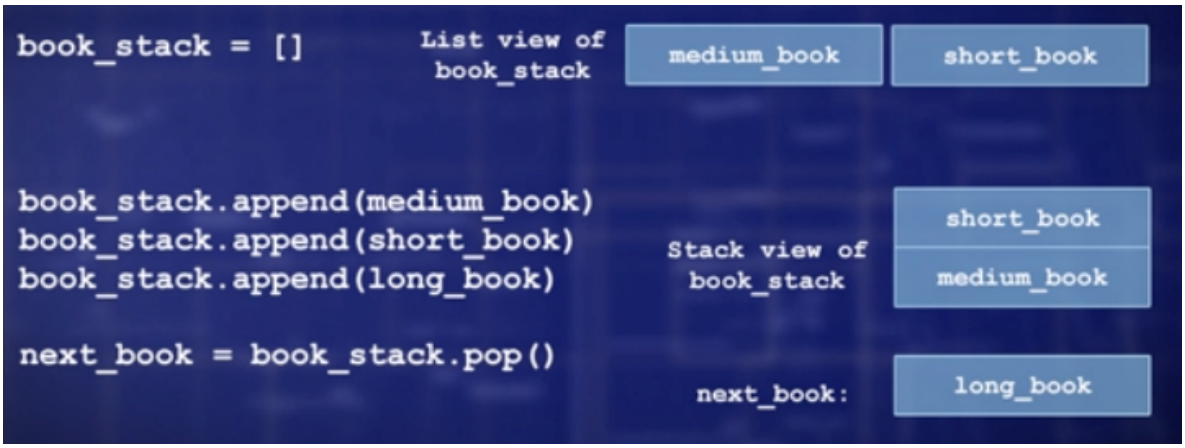


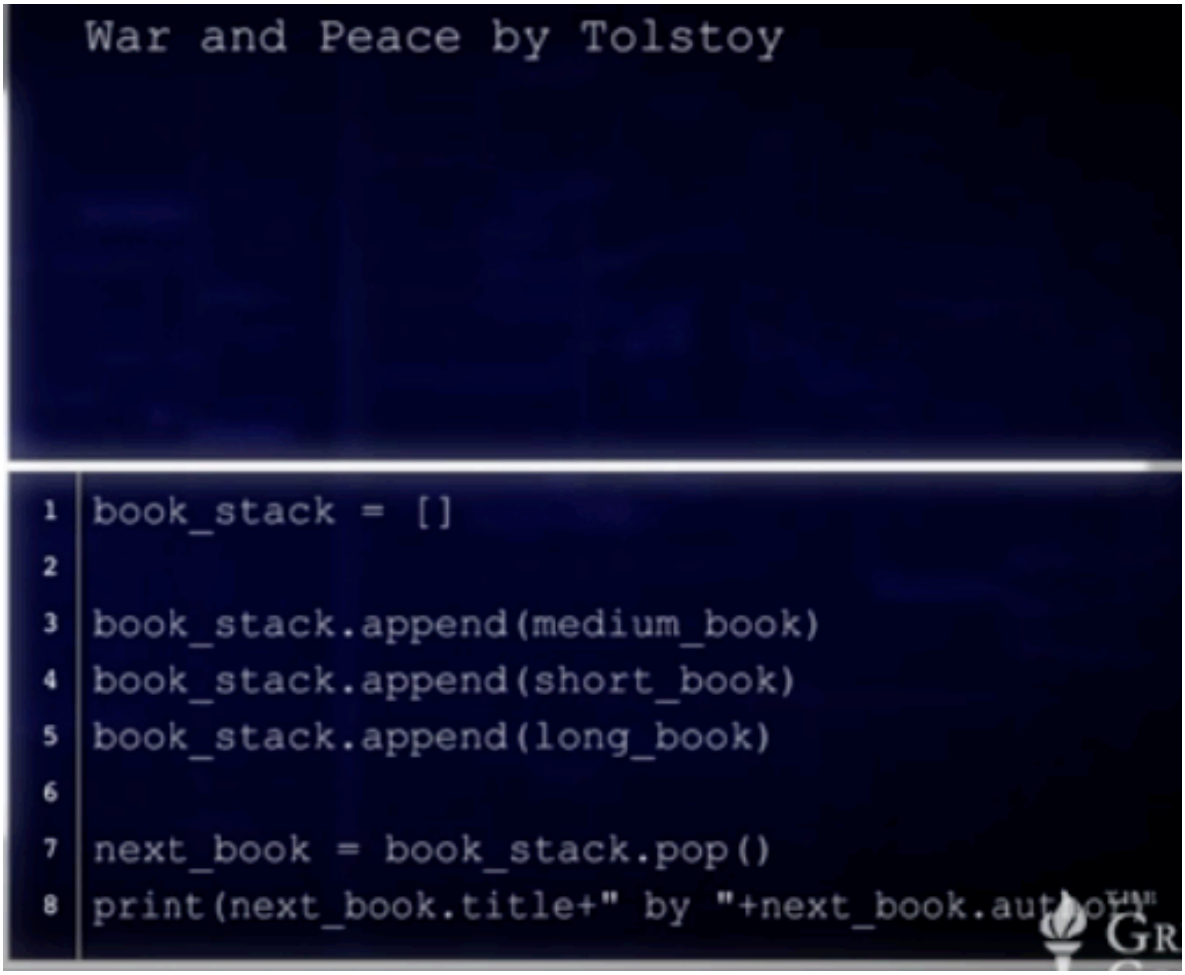

```
1  class book:
2          title = ""
3          author = ""
4
5  long_book = Book()
6  long_book.title = "War and Peace"
7  long_book.author = "Tolstoy"
8
9  medium_book = Book()
10 medium_book.title = "Book of Armaments"
11 medium_book.author = "Maynard"
12
13 short_book = Book()
14 short_book.title = "Vegetables I Like"
15 short_book.author = "John Keyser"
16
```

Stacks

- Represented using a list
- First book in the list is on the bottom
- Last book in the list is on the top
- "Pushing" a book on to the stack requires using the append command on the list

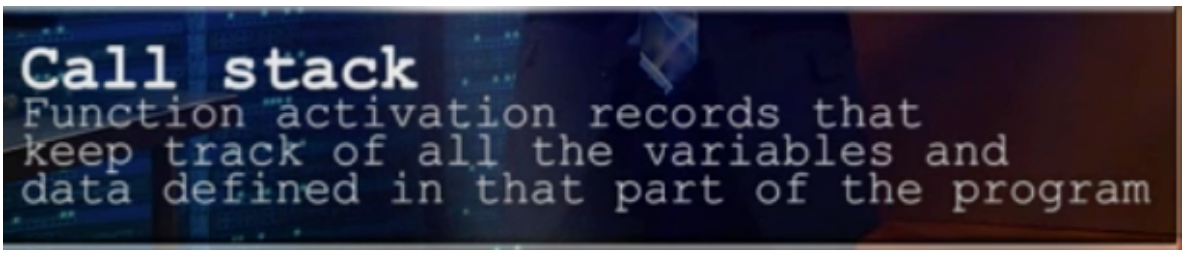



book_stack = []
List view of book_stack
medium_book
short_book
book_stack.append(medium_book)
book_stack.append(short_book)
book_stack.append(long_book)
Stack view of book_stack
short_book
medium_book
next_book = book_stack.pop()
next_book:
long_book


War and Peace by Tolstoy
1 book_stack = []
2
3 book_stack.append(medium_book)
4 book_stack.append(short_book)
5 book_stack.append(long_book)
6
7 next_book = book_stack.pop()
8 print(next_book.title+" by "+next_book.aut

```
class Stack:
    _stack = []
    def push(self, item):
        self._stack.append(item)
    def pop (self):
        return self._stack.pop()

#a stack is "last in, first out"
```

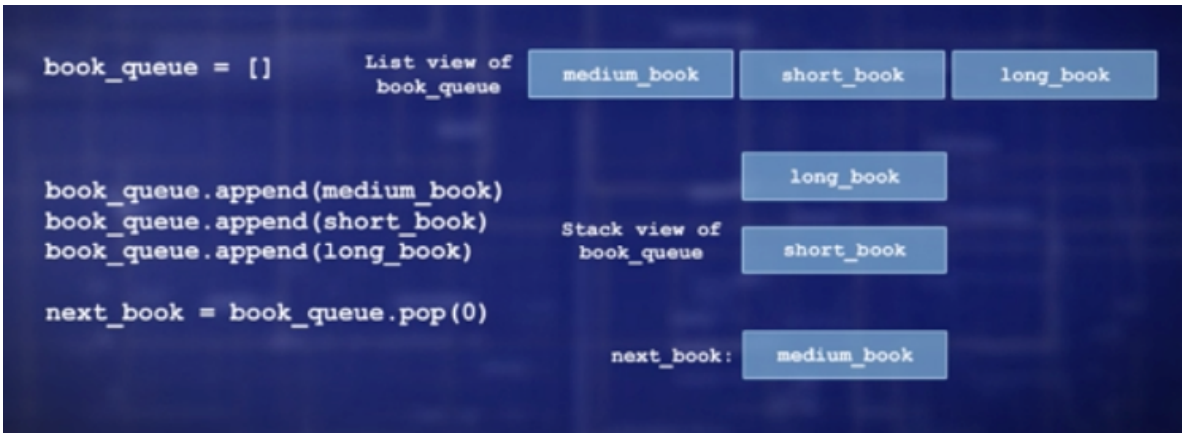
```
class Stack:
    _stack = []
    def push(self, item):
        self._stack.append(item)
    def pop (self):
        return self._stack.pop()

waste_pile = Stack()

#Setting up solitaire game here

while not(noTurnsLeft()):
    #Get the player's move

    if move=="Draw":
        #Get next three cards and push them onto waste stack
        next_card1 = draw_next_card()
        next_card2 = draw_next_card()
        next_card3 = draw_next_card()
        waste_pile.push(next_card1)
        waste_pile.push(next_card2)
        waste_pile.push(next_card3)

    elif move=="PlayfromWaste":
        #Player wants to play the top card from the waste pile
        current_card = waste_pile.pop()

        #Have player play current_card
```



Queues

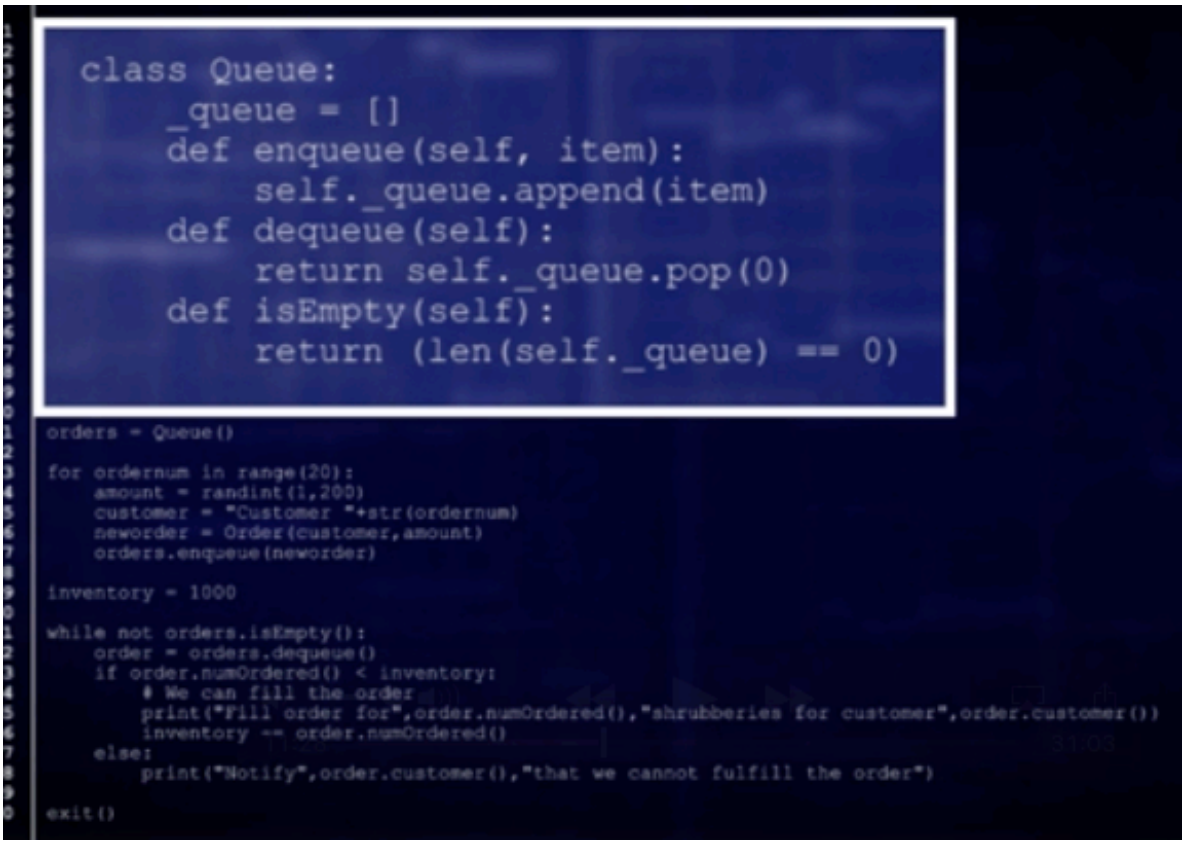
- Order of queue is same as list order
- Can push new objects on to the end of the list using the append command
- Instead of popping from the end of the list, an element is taken off the front


Pop Command
< Takes a parameter, indicating which element gets taken off the list
< If no parameter is given, it defaults to the final element
< To take off the first element, we pass in 0

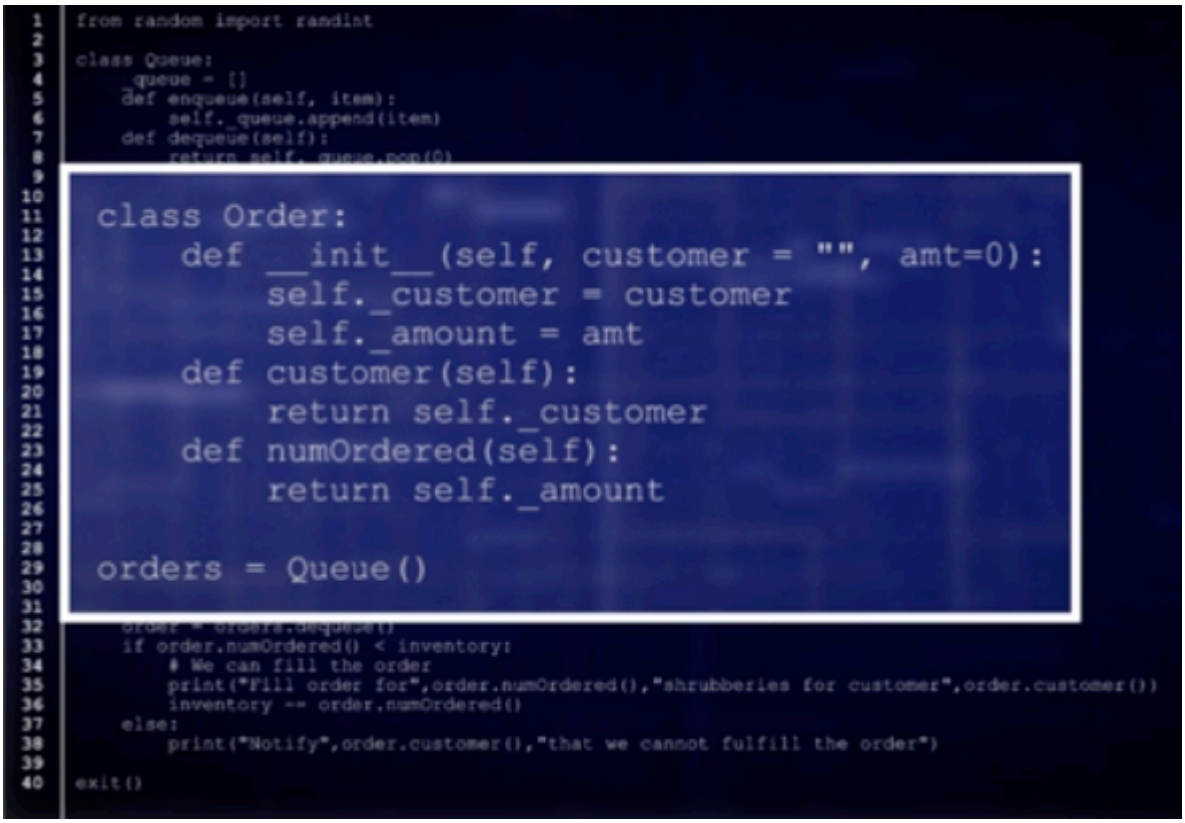

book_queue = []
List view of book_queue
medium_book
short_book
long_book
book_queue.append(medium_book)
book_queue.append(short_book)
book_queue.append(long_book)
next_book = book_queue.pop(0)
Stack view of book_queue
long_book
short_book
next_book:
medium_book

```
1  class Queue:
2      _queue = []
3      def enqueue(self, item):
4          self._queue.append(item)
5      def dequeue(self):
6          return self._queue.pop(0)
7      def isEmpty(self):
8          return (len(self._queue) == 0)
9
```

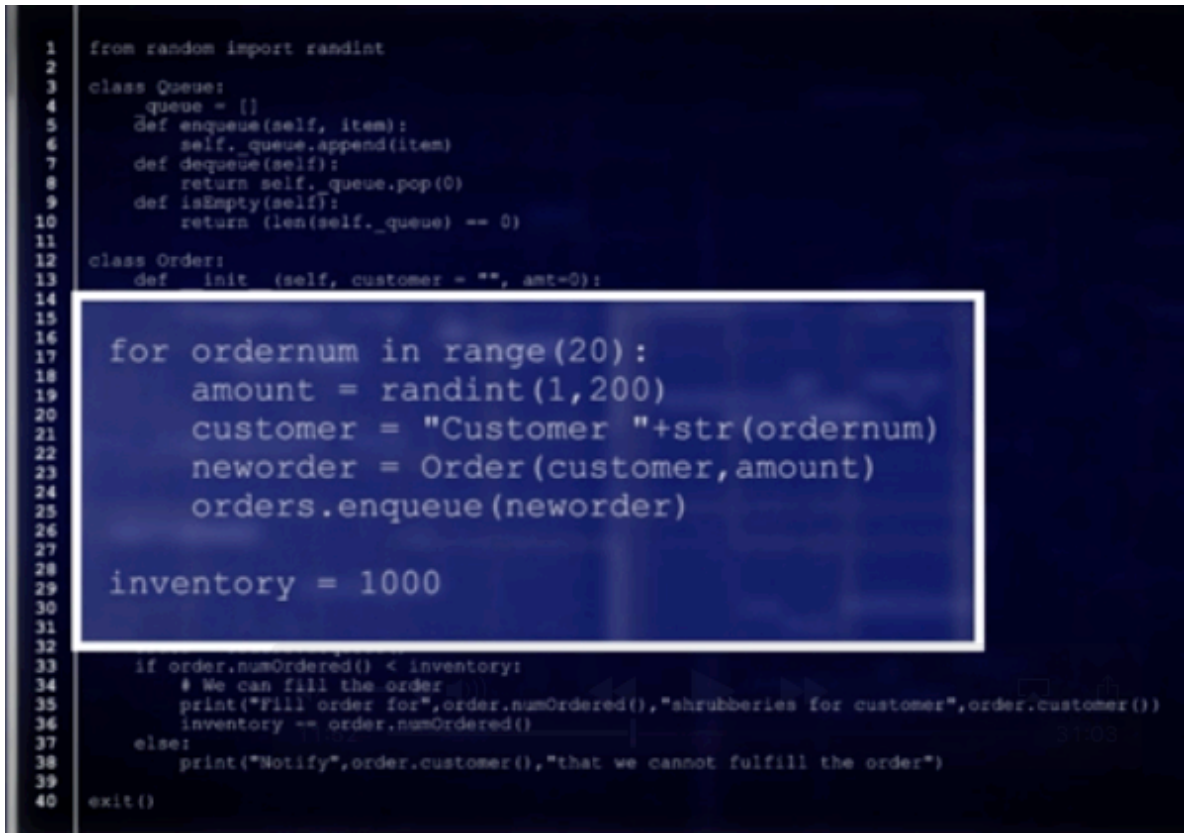
```
1
2
3  class Queue:
4      _queue = []
5      def enqueue(self, item):
6          self._queue.append(item)
7      def dequeue(self):
8          return self._queue.pop(0)
9      def isEmpty(self):
10         return (len(self._queue) == 0)
11
12 class Order:
13     def __init__(self, customer = "", amt=0):
14         self._customer = customer
15         self._amount = amt
16     def customer(self):
17         return self._customer
18     def numOrdered(self):
19         return self._amount
20
21 orders = Queue()
22
23 for ordernum in range(20):
24     amount = randint(1,200)
25     customer = "Customer "+str(ordernum)
26     neworder = Order(customer,amount)
27     orders.enqueue(neworder)
28
29 inventory = 1000
30
31 while not orders.isEmpty():
32     order = orders.dequeue()
33     if order.numOrdered() < inventory:
34         # We can fill the order
35         print("Fill order for",order.numOrdered(),"shrubberies for customer",order.customer())
36         inventory -= order.numOrdered()
37     else:
38         print("Notify",order.customer(),"that we cannot fulfill the order")
39
40 exit()
```



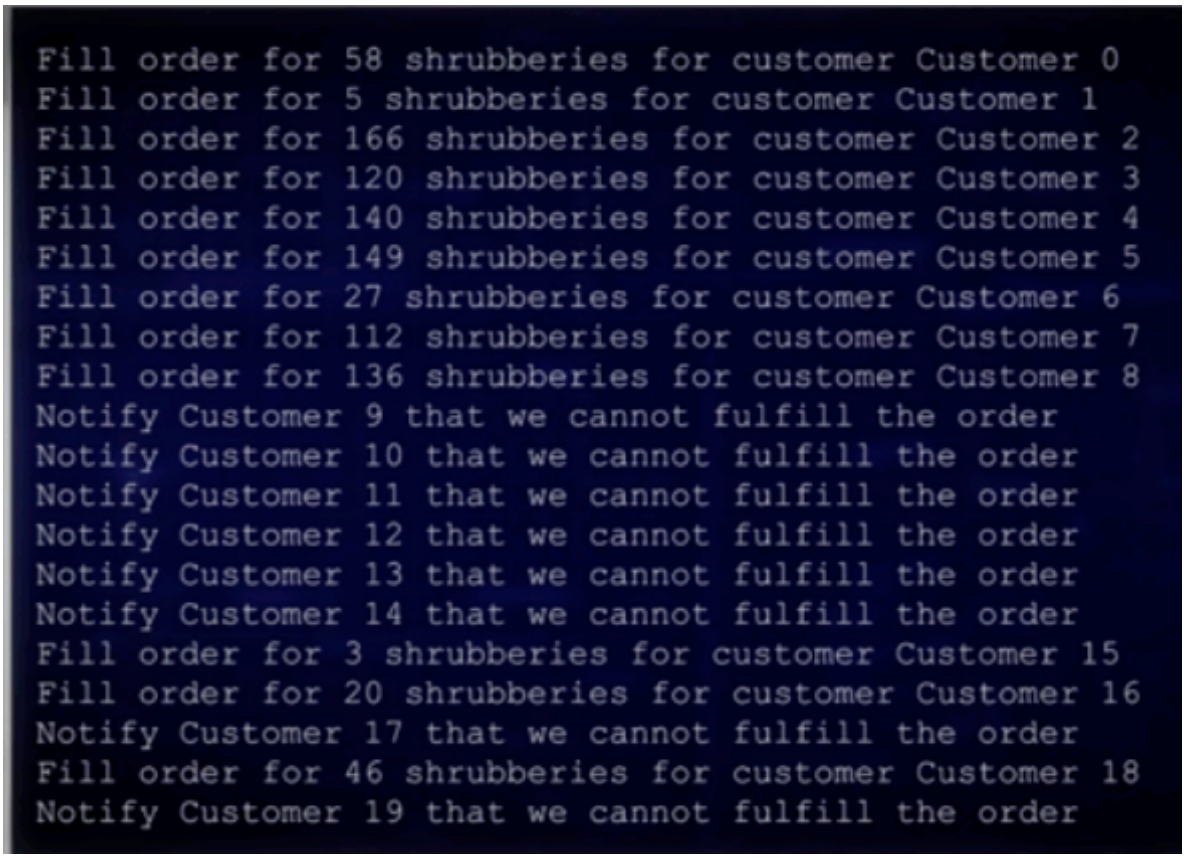
```
class Queue:
    _queue = []
    def enqueue(self, item):
        self._queue.append(item)
    def dequeue(self):
        return self._queue.pop(0)
    def isEmpty(self):
        return (len(self._queue) == 0)

orders = Queue()

for ordernum in range(20):
    amount = randint(1,200)
    customer = "Customer "+str(ordernum)
    neworder = Order(customer,amount)
    orders.enqueue(neworder)

inventory = 1000

while not orders.isEmpty():
    order = orders.dequeue()
    if order.numOrdered() < inventory:
        # We can fill the order
        print("Fill order for",order.numOrdered(),"shrubberies for customer",order.customer())
        inventory -= order.numOrdered()
    else:
        print("Notify",order.customer(),"that we cannot fulfill the order")

exit()
```



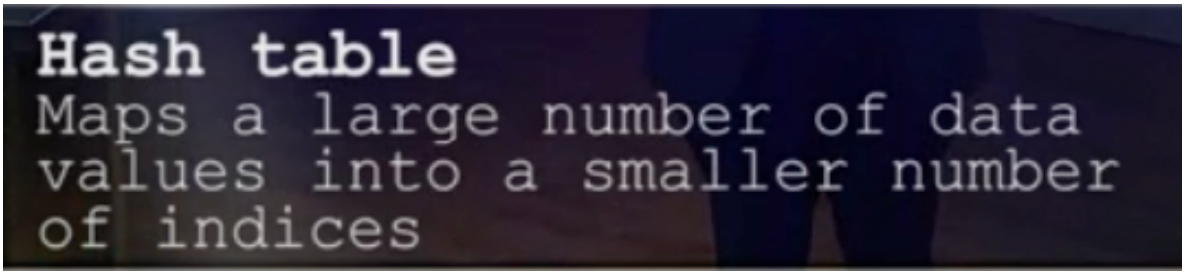
```
from random import randint

class Queue:
    _queue = []
    def enqueue(self, item):
        self._queue.append(item)
    def dequeue(self):
        return self._queue.pop(0)

class Order:
    def __init__(self, customer = "", amt=0):
        self._customer = customer
        self._amount = amt
    def customer(self):
        return self._customer
    def numOrdered(self):
        return self._amount

orders = Queue()

    order = orders.dequeue()
    if order.numOrdered() < inventory:
        # We can fill the order
        print("Fill order for",order.numOrdered(),"shrubberies for customer",order.customer())
        inventory -= order.numOrdered()
    else:
        print("Notify",order.customer(),"that we cannot fulfill the order")

exit()
```

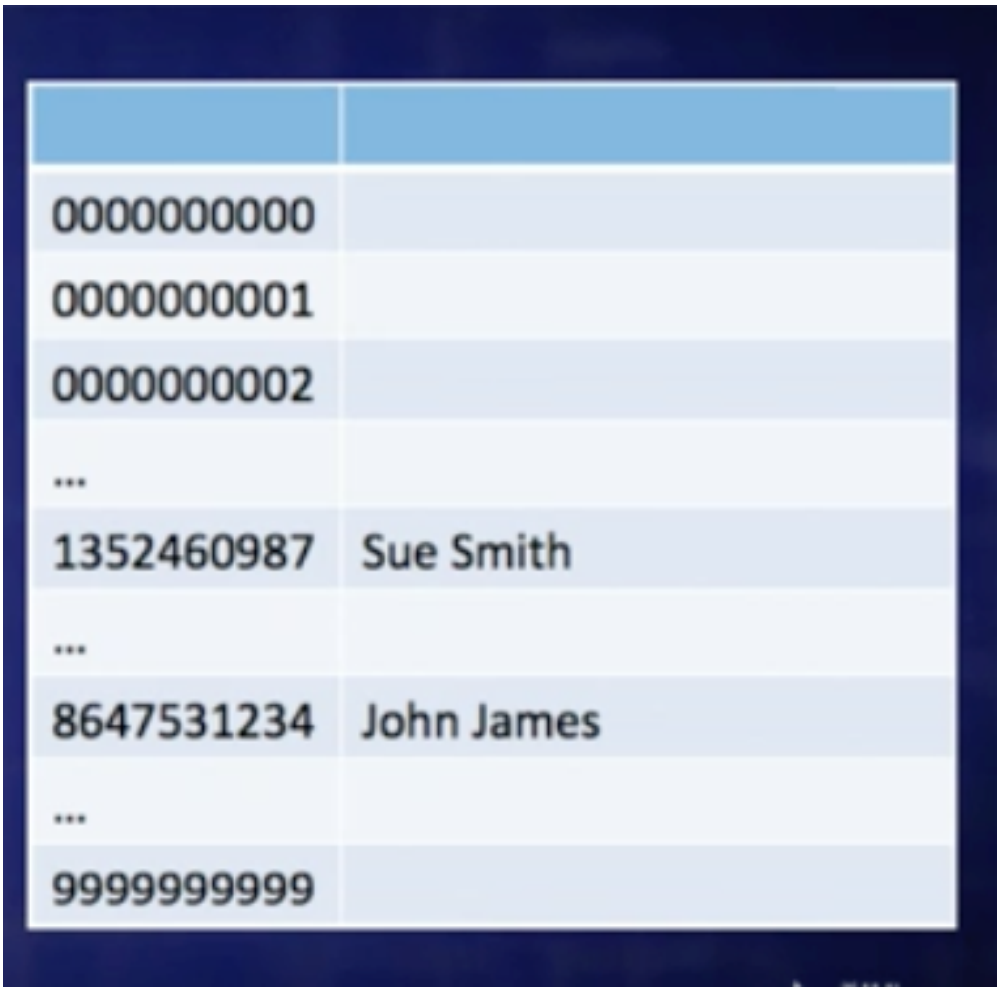

from random import randint
class Queue:
queue = []
def enqueue(self, item):
self._queue.append(item)
def dequeue(self):
return self._queue.pop(0)
def isEmpty(self):
return (len(self._queue) == 0)
class Order:
def __init__(self, customer = "", amt=0):
for ordernum in range(20):
amount = randint(1,200)
customer = "Customer "+str(ordernum)
neworder = Order(customer,amount)
orders.enqueue(neworder)
inventory = 1000
if order.numOrdered() < inventory:
# We can fill the order
print("Fill order for",order.numOrdered(),"shrubberies for customer",order.customer())
inventory -= order.numOrdered()
else:
print("Notify",order.customer(),"that we cannot fulfill the order")
exit()


from random import randint
class Queue:
queue = []
def enqueue(self, item):
self._queue.append(item)
def dequeue(self):
return self._queue.pop(0)
def isEmpty(self):
return (len(self._queue) == 0)
class Order:
def __init__(self, customer = "", amt=0):
self._customer = customer
self._amount = amt
def customer(self):
return self._customer
def numOrdered(self):
return self._amount
orders = Queue()
while not orders.isEmpty():
order = orders.dequeue()
if order.numOrdered() < inventory:
# We can fill the order
print("Fill order for",order.numOrdered(),"shrubberies for customer",order.customer())
inventory -= order.numOrdered()
else:
print("Notify",order.customer(),"that we cannot fulfill the order")
exit()

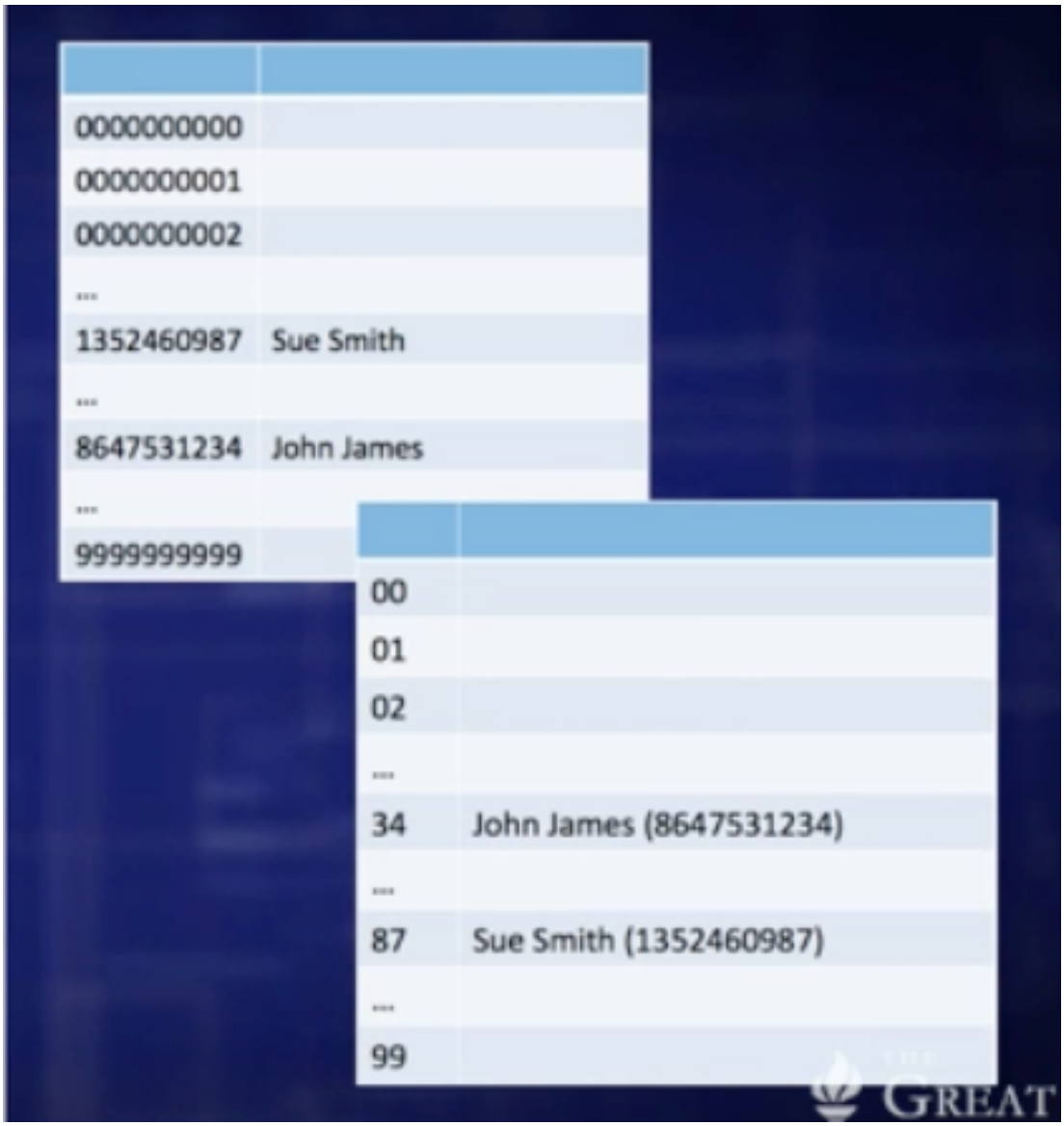

Fill order for 58 shrubberies for customer Customer 0
Fill order for 5 shrubberies for customer Customer 1
Fill order for 166 shrubberies for customer Customer 2
Fill order for 120 shrubberies for customer Customer 3
Fill order for 140 shrubberies for customer Customer 4
Fill order for 149 shrubberies for customer Customer 5
Fill order for 27 shrubberies for customer Customer 6
Fill order for 112 shrubberies for customer Customer 7
Fill order for 136 shrubberies for customer Customer 8
Notify Customer 9 that we cannot fulfill the order
Notify Customer 10 that we cannot fulfill the order
Notify Customer 11 that we cannot fulfill the order
Notify Customer 12 that we cannot fulfill the order
Notify Customer 13 that we cannot fulfill the order
Notify Customer 14 that we cannot fulfill the order
Fill order for 3 shrubberies for customer Customer 15
Fill order for 20 shrubberies for customer Customer 16
Notify Customer 17 that we cannot fulfill the order
Fill order for 46 shrubberies for customer Customer 18
Notify Customer 19 that we cannot fulfill the order

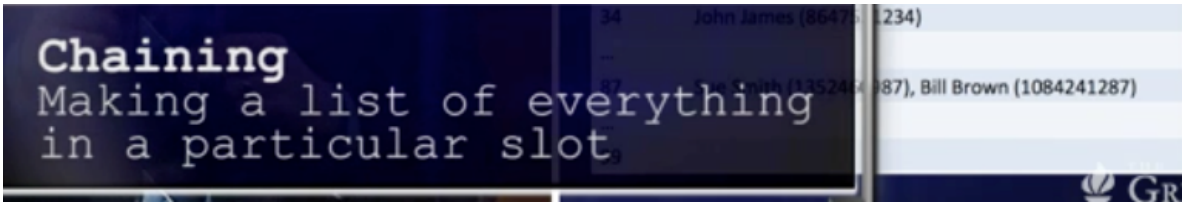

Buffer
Queue of software events, like mouse movements or a keyboard key being pressed


Hash table
Maps a large number of data values into a smaller number of indices



0000000000	
0000000001	
0000000002	
...	
1352460987	Sue Smith
...	
8647531234	John James
...	
9999999999	


0000000000
0000000001
0000000002
...
1352460987 Sue Smith
...
8647531234 John James
...
9999999999
00
01
02
...
34 John James (8647531234)
...
87 Sue Smith (1352460987)
...
99
GREAT


Chaining
Making a list of everything
in a particular slot
234)
87), Bill Brown (1084241287)

Nickname	Person
SuperStar	Sue Smith
CowboysFan	Bill Brown
JJwins	John James

00	
01	
02	
...	

Hash function

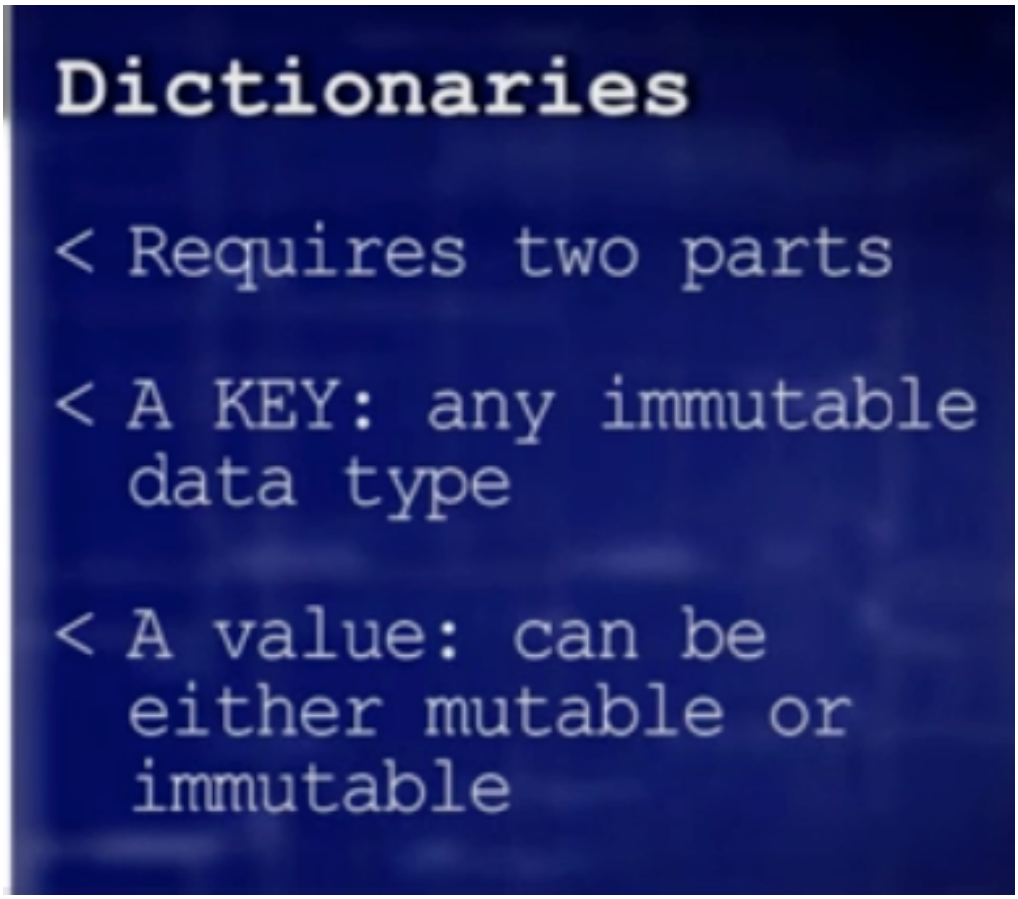
Function to convert a key phrase into a number to index into an array (list)

Nickname	Person
SuperStar	Sue Smith
CowboysFan	Bill Brown
JJwins	John James

00	
01	Bill Brown (CowboysFan)
02	
...	
71	Sue Smith (SuperStar)
...	
90	John James (JJwins)
...	
99	


Bill Brown
1 my_dictionary = {}
2 nicknames = {"Superstar" : "Sue Smith",
3 "CowboysFan" : "Bill Brown",
4 "JJwins" : "John James"}
5
6 print(nicknames["CowboysFan"])
7


Bill Brown
1 nicknames = []
2 nicknames["Superstar"] = "Sue Smith"
3 nicknames["CowboysFan"] = "Bill Brown"
4 nicknames["JJwins"] = "John James"
5
6 print(nicknames["CowboysFan"])


Dictionaries
< Requires two parts
< A KEY: any immutable data type
< A value: can be either mutable or immutable

```
True
False
```

```
1  nicknames = {}
2
3  nicknames["Superstar"] = "Sue Smith"
4  nicknames["CowboysFan"] = "Bill Brown"
5  nicknames["JJwins"] = "John James"
6
7  del nicknames["Superstar"]
8
9  print("CowboysFan" in nicknames)
10 print("Superstar" in nicknames)
```

```
True
False
```

```
1  nicknames = {}
2
3  nicknames["Superstar"] = "Sue Smith"
4  nicknames["CowboysFan"] = "Bill Brown"
5  nicknames["JJwins"] = "John James"
6
7  del nicknames["Superstar"]
8
9  print("CowboysFan" in nicknames)
10 print("Superstar" in nicknames)
```

```
The nickname for Sue Smith is Superstar
The nickname for John James is JJwins
The nickname for Bill Brown is CowboysFan
```

```
1  nicknames = {}
2
3  nicknames["Superstar"] = "Sue Smith"
4  nicknames["CowboysFan"] = "Bill Brown"
5  nicknames["JJwins"] = "John James"
6
7  for nickname in nicknames:
8      print("The nickname for "+nicknames[nickname]+" is "+nickname)
```

```
1  passwords = {"John" : "123456", "Sue" : "PaSsWoRd", "Bill" : "G9.Kf-21.-fe8ilfb" }
2
3  failed_attempts = 0
4  verified = False
5
6  while (not verified):
7      username = input("What is your username? ")
8      password = input("What is your password? ")
9      if (username in passwords) and (passwords[username] == password):
10         print ("Welcome!")
11         verified = True
12     else:
13         print ("Invalid username/password combination")
14         failed_attempts += 1
15         if failed_attempts > 2:
16             print("Too many incorrect attempts.  Goodbye")
17             exit()
```

Set

Collection of items stored using a hash table-does mathematical set operations-checks membership very quickly

```
ingredients = {'flour','sugar','eggs','milk'}
```

```
Yes!
```

```
1  people = {'John', 'Sue', 'Bill'}
2
3  if 'John' in people:
4      print("Yes!")
5  else:
6      print("No!")
7
```

```
Yes!
```

```
1 people = set(['John', 'Sue', 'Bill'])
2
3 if 'John' in people:
4     print("Yes!")
5 else:
6     print("No!")
```

```
{'Fred', 'Eric', 'Sue'}
{'Fred', 'Eric', 'Sue', 'Kathy'}
{'Eric', 'Sue', 'Kathy'}
```

```
1 work_friends = {'Sue', 'Eric', 'Fred'}
2
3 print(work_friends)
4 work_friends.add('Kathy')
5 print(work_friends)
6 work_friends.remove('Fred')
7 print(work_friends)
```

```
{'John', 'Bill'}
{'Fred', 'Bill', 'Eric', 'John', 'Sue'}
{'Sue'}
{'Eric', 'Fred', 'Bill', 'John'}
```

```
1  neighborhood_friends = set(['John', 'Sue', 'Bill'])
2  work_friends = {'Sue', 'Eric', 'Fred'}
3
4  print(neighborhood_friends - work_friends)
5  print(neighborhood_friends | work_friends)
6  print(neighborhood_friends & work_friends)
7  print(neighborhood_friends ^ work_friends)
8
```

```
Here is the set of items you need to buy:
{'Yeast', 'Eggs', 'Flour', 'Peppers'}
```

```
1  class recipe:
2      name=''
3      ingredients = []
4      def __init__(self, name, ingredients):
5          self.name = name
6          self.ingredients = ingredients
7
8  dish1 = recipe("Omlette", ['Eggs', 'Tomatoes', 'Onions', 'Peppers'])
9  dish2 = recipe("Bread", ['Flour', 'Yeast'])
10 dish3 = recipe("Cake", ['Eggs', 'Flour', 'Sugar', 'Butter'])
11 dishes_to_fix = [dish1, dish2, dish3]
12
13 shopping_list = set()
14 for dish in dishes_to_fix:
15     ingredients = set(dish.ingredients)
16     shopping_list = shopping_list | ingredients
17
18 ingredients_on_hand = {'Onions', 'Butter', 'Milk', 'Honey', 'Oatmeal', 'Sugar', 'Tomatoes'}
19 shopping_list -= ingredients_on_hand
20
21 print("Here is the set of items you need to buy:")
22 print(shopping_list)
```

```
Here is the set of items you need to buy:
{'Yeast', 'Eggs', 'Flour', 'Peppers'}
```

```
class recipe:
    name=''
    ingredients = []
    def __init__(self, name, ingredients):
        self.name = name
        self.ingredients = ingredients

for dish in dishes_to_fix:
    ingredients = set(dish.ingredients)
    shopping_list = shopping_list | ingredients

ingredients_on_hand = {'Onions', 'Butter', 'Milk', 'Honey', 'Oatmeal', 'Sugar', 'Tomatoes'}
shopping_list -= ingredients_on_hand

print("Here is the set of items you need to buy:")
print(shopping_list)
```


```
Here is the set of items you need to buy:
{'Yeast', 'Eggs', 'Flour', 'Peppers'}
```


```
class recipe:
    name=''
    ingredients = []

dish1 = recipe("Omlette", ['Eggs', 'Tomatoes', 'Onions', 'Peppers'])
dish2 = recipe("Bread", ['Flour', 'Yeast'])
dish3 = recipe("Cake", ['Eggs', 'Flour', 'Sugar', 'Butter'])
dishes_to_fix = [dish1, dish2, dish3]

shopping_list = set()
for dish in dishes_to_fix:
    ingredients = set(dish.ingredients)
    shopping_list = shopping_list | ingredients

ingredients_on_hand = {'Onions', 'Butter', 'Milk', 'Honey', 'Oatmeal', 'Sugar', 'Tomatoes'}
shopping_list -= ingredients_on_hand

print("Here is the set of items you need to buy:")
print(shopping_list)
```

Here is the set of items you need to buy:

{'Yeast', 'Eggs', 'Flour', 'Peppers'}

```
class recipe:
    name=''
    ingredients = []
    def __init__(self, name, ingredients):
        self.name = name
        self.ingredients = ingredients

dish1 = recipe("Omlette", ['Eggs', 'Tomatoes', 'Onions', 'Peppers'])
dish2 = recipe("Bread", ['Flour', 'Yeast'])

shopping_list = set()
for dish in dishes_to_fix:
    ingredients = set(dish.ingredients)
    shopping_list = shopping_list | ingredients

print("Here is the set of items you need to buy:")
print(shopping_list)
```

Here is the set of items you need to buy:

{'Yeast', 'Eggs', 'Flour', 'Peppers'}

```
class recipe:
    name=''
    ingredients = []
    def __init__(self, name, ingredients):
        self.name = name
        self.ingredients = ingredients

dish1 = recipe("Omlette", ['Eggs', 'Tomatoes', 'Onions', 'Peppers'])
dish2 = recipe("Bread", ['Flour', 'Yeast'])
dish3 = recipe("Cake", ['Eggs', 'Flour', 'Sugar', 'Butter'])
dishes_to_fix = [dish1, dish2, dish3]

shopping_list = set()

ingredients_on_hand = {'Onions', 'Butter', 'Milk', 'Honey', 'Oatmeal', 'Sugar', 'Tomatoes'}
shopping_list -= ingredients_on_hand

print("Here is the set of items you need to buy:")
print(shopping_list)
```

Linear, Sequential (using list)

< stacks
< queues

Nonlinear, Non-sequential (using hash table)

< dictionaries
< sets