⌄ Iterators

- An iterator is an object that contains a countable number of values.

- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.

- Lists, tuples, dictionaries, and sets are all iterable objects.

- They are iterable containers which you can get an iterator from.

All these objects have a iter() method which is used to get an iterator

Even strings are iterable objects, and can return an iterator. Strings are also iterable objects, containing a sequence of characters

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)
print(myit)

print(next(myit))
print(next(myit))
print(next(myit))
```

⇥   `<tuple_iterator object at 0x7977eb37e290>`
     `apple`
     `banana`
     `cherry`

```
mylist = ["apple", "banana", "cherry"]
myit = iter(mylist)
print(myit)

print(next(myit))
print(next(myit))
print(next(myit))
```

⇥   `<list_iterator object at 0x7977eb37eaa0>`
     `apple`
     `banana`
     `cherry`

```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

⇥   `b`
     `a`
     `n`
     `a`
     `n`
     `a`

```
mydict = {1:'c',2:'python'}
myit = iter(mydict.items())

print(next(myit))
print(next(myit))
```

⇥   `(1, 'c')`
     `(2, 'python')`

```
mylist = ["apple", "banana", "cherry"]
myit = iter(mylist)
print(myit)

for i in range(len(mylist)):
    print(next(myit))
```

```
<list_iterator object at 0x7977eb37efe0>
    apple
    banana
    cherry
```

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)
print(myit)

for i in range(len(mytuple)):
    print(next(myit))
```

```
<tuple_iterator object at 0x7977eb37fe80>
    apple
    banana
    cherry
```

```
mystr = "apple"
myit = iter(mystr)
print(myit)

for i in range(len(mystr)):
    print(next(myit))
```

```
<str_ascii_iterator object at 0x7977eb3448e0>
    a
    p
    p
    l
    e
```

Polymorphism

The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

- Polymorphism in functions: built in and user defined
- Polymorphism in classes

Function Polymorphism An example of a Python function that can be used on different objects is the len() function.

Polymorphism in functions

⌄ String

For strings len() returns the number of characters:

```
x = "Hello World!"

print(len(x))
```

```
12
```

⌄ Tuple

For tuples len() returns the number of items in the tuple:

```
mytuple = ("apple", "banana", "cherry")

print(len(mytuple))
```

```
3
```

Dictionary

For dictionaries len() returns the number of key/value pairs in the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

print(len(thisdict))
```

⇥ 3

Polymorphism in classes

Polymorphism is often used in Class methods, where we can have multiple classes with the same method name.

For example, say we have three classes: Car, Boat, and Plane, and they all have a method called move():

```
class Car:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Drive!")

class Boat:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Sail!")

class Plane:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang")        #Create a Car object
boat1 = Boat("Ibiza", "Touring 20")  #Create a Boat object
plane1 = Plane("Boeing", "747")      #Create a Plane object

for x in (car1, boat1, plane1):
  x.move()
```

⇥ Drive!
   Sail!
   Fly!

Polymorphism in inheritance

What about classes with child classes with the same name? Can we use polymorphism there?

Yes. If we use the example above and make a parent class called Vehicle, and make Car, Boat, Plane child classes of Vehicle, the child classes inherits the Vehicle methods, but can override them:

Create a class called Vehicle and make Car, Boat, Plane child classes of Vehicle

```
class Vehicle:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model
```

```
  def move(self):
    print("Move!")

class Car(Vehicle):
  pass

class Boat(Vehicle):
  def move(self):
    print("Sail!")

class Plane(Vehicle):
  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang")        #Create a Car object
boat1 = Boat("Ibiza", "Touring 20")  #Create a Boat object
plane1 = Plane("Boeing", "747")      #Create a Plane object

for x in (car1, boat1, plane1):
  print(x.brand)
  print(x.model)
  x.move()
```

```
⇥  Ford
   Mustang
   Move!
   Ibiza
   Touring 20
   Sail!
   Boeing
   747
   Fly!
```

Child classes inherits the properties and methods from the parent class.

In the example above you can see that the Car class is empty, but it inherits brand, model, and move() from Vehicle.

The Boat and Plane classes also inherit brand, model, and move() from Vehicle, but they both override the move() method.

Because of polymorphism we can execute the same method for all classes.

∨ Super function to access parent class data

super()

```
class Vehicle:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Move!")

class Plane(Vehicle):

  def move(self):
    super().move()
    print("Welcome to ", self.brand, self.model)
    print("Fly!")

x = Plane("Boeing", "747")      #Create a Plane object
print(x.brand)
print(x.model)
x.move()
```

```
⇥  Boeing
   747
   Move!
   Welcome to  Boeing 747
   Fly!
```

Start coding or generate with AI.