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!
```

Operator Overloading

```
# 1. Add two objects using add operator

class A:
    def __init__(self, a):
        self.a = a

    # adding two objects
    def __add__(self, o):
        return self.a + o.a
ob1 = A(1)
ob2 = A(2)
ob3 = A("Hello")
ob4 = A(' Python')

print(ob1 + ob2)
print(ob3 + ob4)
# Actual working when Binary Operator is used.
print(A.__add__(ob1 , ob2))
print(A.__add__(ob3,ob4))
#And can also be Understand as :
print(ob1.__add__(ob2))
print(ob3.__add__(ob4))
```

```
3
Hello Python
3
Hello Python
3
Hello Python
```

```
# 1. Add two objects using add operator

class A:
    def __init__(self, a):
        self.a = a

    # adding two objects
    def __add__(self, o):
        return self.a * o.a
ob1 = A(4)
ob2 = A(2)
ob3 = A("Hello ")
ob4 = A(3)

print(ob1 + ob2)
print(ob3 + ob4)
# Actual working when Binary Operator is used.
print(A.__add__(ob1 , ob2))
print(A.__add__(ob3,ob4))
#And can also be Understand as :
print(ob1.__add__(ob2))
print(ob3.__add__(ob4))
```

```
8
Hello Hello Hello
8
Hello Hello Hello
8
Hello Hello Hello
```

```
class A:
    def __init__(self, a):
        self.a = a

    def __add__(self, other):
        return self.a + other.a

    def __mul__(self, other):
        return self.a * other.a

obj1 = A(2)
obj2 = A(3)
```

```
print(obj1 + obj2)   # Output: 5
print(obj1 * obj2)   # Output: 6
```

```
5
6
```

```
class Number:
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return self.value + other.value

    def __sub__(self, other):
        return self.value - other.value

    def __mul__(self, other):
        return self.value * other.value

    def __truediv__(self, other):
        return self.value / other.value

    def __floordiv__(self, other):
        return self.value // other.value

    def __mod__(self, other):
        return self.value % other.value

    def __pow__(self, other):
        return self.value ** other.value


n1 = Number(10)
n2 = Number(3)

print("Addition:", n1 + n2)
print("Subtraction:", n1 - n2)
print("Multiplication:", n1 * n2)
print("Division:", n1 / n2)
print("Floor Division:", n1 // n2)
print("Modulus:", n1 % n2)
print("Power:", n1 ** n2)
```

```
Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3.3333333333333335
Floor Division: 3
Modulus: 1
Power: 1000
```

```
class Value:
    def __init__(self, x):
        self.x = x

    def __neg__(self):
        return -self.x

    def __pos__(self):
        return +self.x

    def __invert__(self):
        return ~self.x


v = Value(5)
print("Negation:", -v)
print("Unary Plus:", +v)
print("Bitwise NOT:", ~v)
```

```
Negation: -5
Unary Plus: 5
Bitwise NOT: -6
```

```python
class Compare:
    def __init__(self, x):
        self.x = x

    def __lt__(self, other):
        return self.x < other.x

    def __le__(self, other):
        return self.x <= other.x

    def __eq__(self, other):
        return self.x == other.x

    def __ne__(self, other):
        return self.x != other.x

    def __gt__(self, other):
        return self.x > other.x

    def __ge__(self, other):
        return self.x >= other.x


c1 = Compare(5)
c2 = Compare(10)

print("Less than:", c1 < c2)
print("Less than or equal:", c1 <= c2)
print("Equal to:", c1 == c2)
print("Not equal to:", c1 != c2)
print("Greater than:", c1 > c2)
print("Greater than or equal:", c1 >= c2)
```

```
Less than: True
Less than or equal: True
Equal to: False
Not equal to: True
Greater than: False
Greater than or equal: False
```

```python
class Bitwise:
    def __init__(self, num):
        self.num = num

    def __and__(self, other):
        return self.num & other.num

    def __or__(self, other):
        return self.num | other.num

    def __xor__(self, other):
        return self.num ^ other.num

    def __lshift__(self, other):
        return self.num << other.num

    def __rshift__(self, other):
        return self.num >> other.num


b1 = Bitwise(10)  # 1010
b2 = Bitwise(2)   # 0010

print("AND:", b1 & b2)
print("OR:", b1 | b2)
print("XOR:", b1 ^ b2)
print("Left Shift:", b1 << b2)
print("Right Shift:", b1 >> b2)
```

```
AND: 2
OR: 10
XOR: 8
Left Shift: 40
Right Shift: 2
```

```python
class Count:
    def __init__(self, value):
        self.value = value

    def __iadd__(self, other):
        self.value += other.value
        return self  # Return self, not self.value

    def __isub__(self, other):
        self.value -= other.value
        return self

    def __imul__(self, other):
        self.value *= other.value
        return self


c1 = Count(5)
c2 = Count(3)

print("Initial:", c1.value)
c1 += c2
print("After += :", c1.value)
c1 -= c2
print("After -= :", c1.value)
c1 *= c2
print("After *= :", c1.value)
```

```
Initial: 5
After += : 8
After -= : 5
After *= : 15
```

```python
class MyList:
    def __init__(self, items):
        self.items = items

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

    def __str__(self):
        return f"MyList: {self.items}"

    def __repr__(self):
        return f"MyList({self.items})"

    def __abs__(self):
        return [abs(i) for i in self.items]

    def __bool__(self):
        return bool(self.items)


ml = MyList([1, -2, 3])
print("Length:", len(ml))
print("String:", str(ml))
print("Representation:", repr(ml))
print("Absolute values:", abs(ml))
print("Is non-empty?:", bool(ml))

empty = MyList([])
print("Is empty?:", bool(empty))
```

```
Length: 3
String: MyList: [1, -2, 3]
Representation: MyList([1, -2, 3])
Absolute values: [1, 2, 3]
Is non-empty?: True
Is empty?: False
```