

# Some Topics

# Nested Functions

- A Function may be defined in other function.

.

# Nested Functions

```
*nested.py - C:/Documents and Settings/admin/Desktop/intro-python/examples/22-some-topics/nested.py
File Edit Format Run Options Window Help

def mult(X, Y):
    def add(A, B):
        C=A
        D=B
        while (D!=0) :
            D=D-1
            C=C+1
        return(C)
    C=0
    D=X
    while (D!=0) :
        D=D-1
        C=add(Y,C)
    return(C)

A = int(input("Enter first number:"))
B = int(input("Enter second number:"))
C=mult(A, B)

print("the result is:",C)

#output
Enter first number:5
Enter second number:4
the result is: 20
```

## Indirect Function Calls

- Functions may be assigned to other names, passed to other functions, embedded in data structures, returned from one function to another, and more, as if they were simple numbers or strings.
- Functions also happen to support a special operation: they can be called by listing arguments in parentheses after a function expression.

# Indirect Function Calls

```
def echo(message):           # Name echo assigned to function object
    print(message)

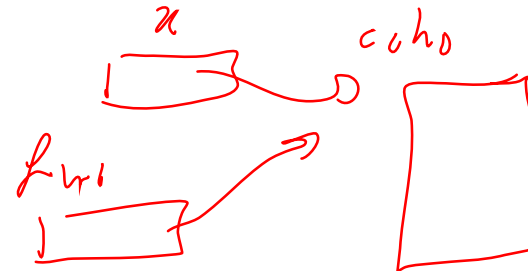
echo('Direct call')          # Call object through original name
x = echo                     # Now x references the function too
x('Indirect call!')         # Call object through name by adding ()

def indirect(func, arg):     # Call the passed-in object by adding ()
    func(arg)

                                # Pass the function to another function
indirect(echo, 'Argument call!')
```

#output

Direct call  
Indirect call!  
Argument call!



# Indirect Function Calls

```
def echo(message):          # Name echo assigned to :
    print(message)

schedule = [ (echo, 'Spam!'), (echo, 'Ham!') ]
for (func, arg) in schedule:
    func(arg)               # Call functions embedded

#output
    Spam!
    Ham!

print('Good bye')
```

# Factory Functions

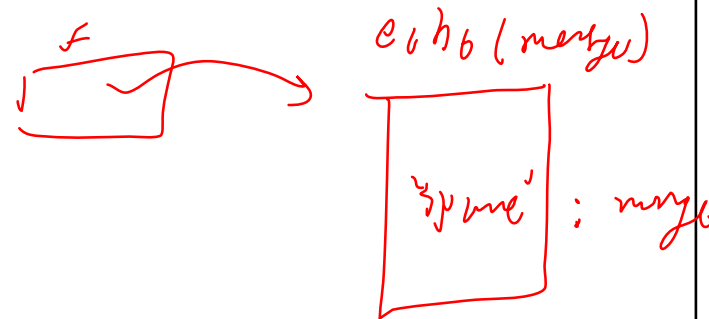
- We may not always know what kind of objects we want to create in advance. Some objects could be created *only* at execution time after a user requests so.
- To deal with this we can use the *factory method* pattern. The idea is to have one function, the factory, that takes an input string and outputs an object. Thus, the factory returns objects.

# Factory Functions

```
def make(label):    # Make a function but don't call it
    def echo(message):
        print(label + ':' + message)
    return echo
```

```
F = make('Spam')    # Label in enclosing scope is retained
F('Ham!')           # Call the function that make returned
F('Eggs!')
```

```
#output
Spam:Ham!
Spam:Eggs!
```





# Factory Functions

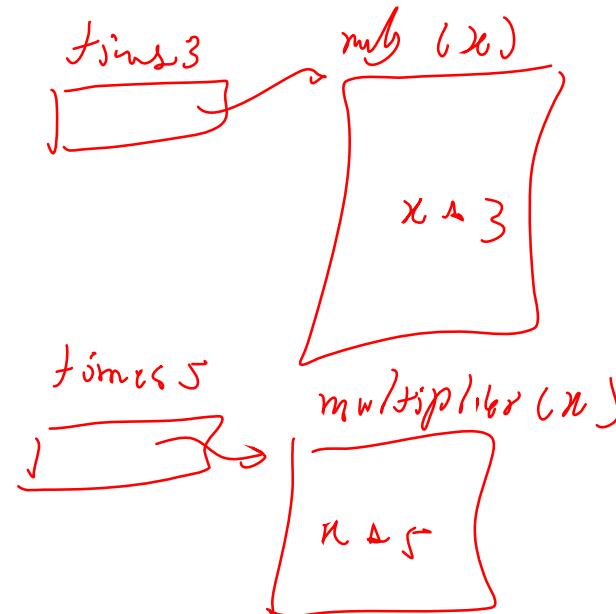
```
def make_multiplier_of(n):  
    def multiplier(x):  
        return x * n  
    return multiplier
```

```
times3 = make_multiplier_of(3)  
times5 = make_multiplier_of(5)
```

```
print(times3(9))  
print(times5(3))  
print(times5(times3(2)))
```

#output:

```
27  
15  
30
```



# Factory Functions

```
def maker(N):  
    def action(X):          # Make and return action  
        return X ** N      # action retains N from enclosing scope  
    return action
```

```
f=maker(2)
```

```
print(f(3))
```

```
print(f(4))
```

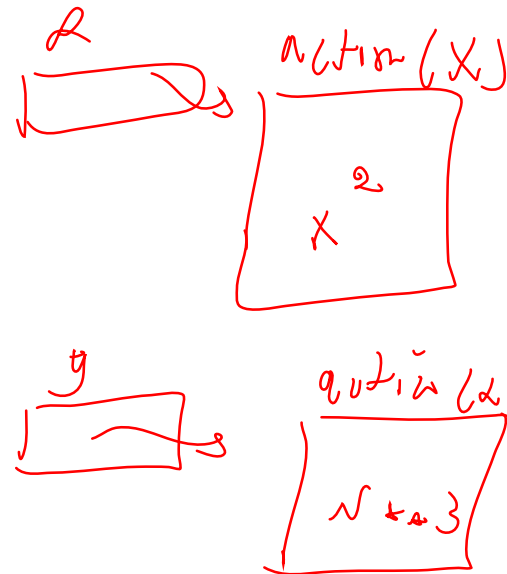
```
g=maker(3)
```

```
print(g(3))
```

```
print(g(4))
```

#output

```
9  
16  
27  
64
```

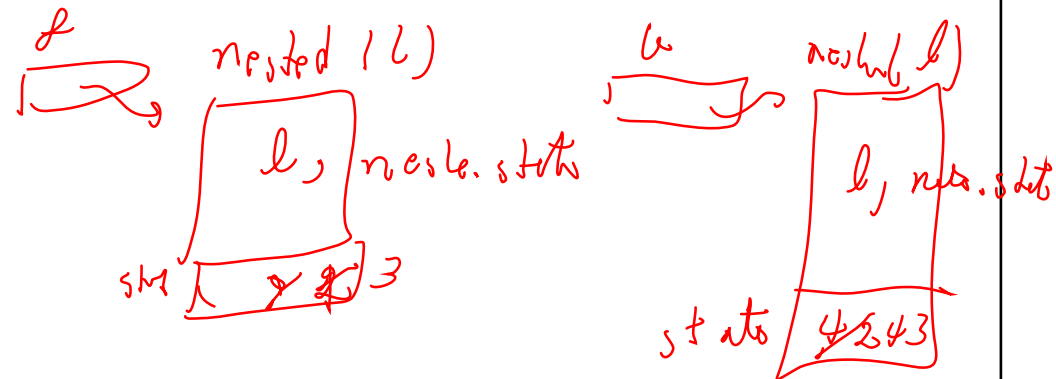


# Factory Functions Dot notation

```
def tester(start):  
    def nested(label):  
        print(label, nested.state) # nested is in enclosing scope  
        nested.state += 1          # Change attr, not nested itself  
    nested.state = start            # Initial state after func defined  
    return nested
```

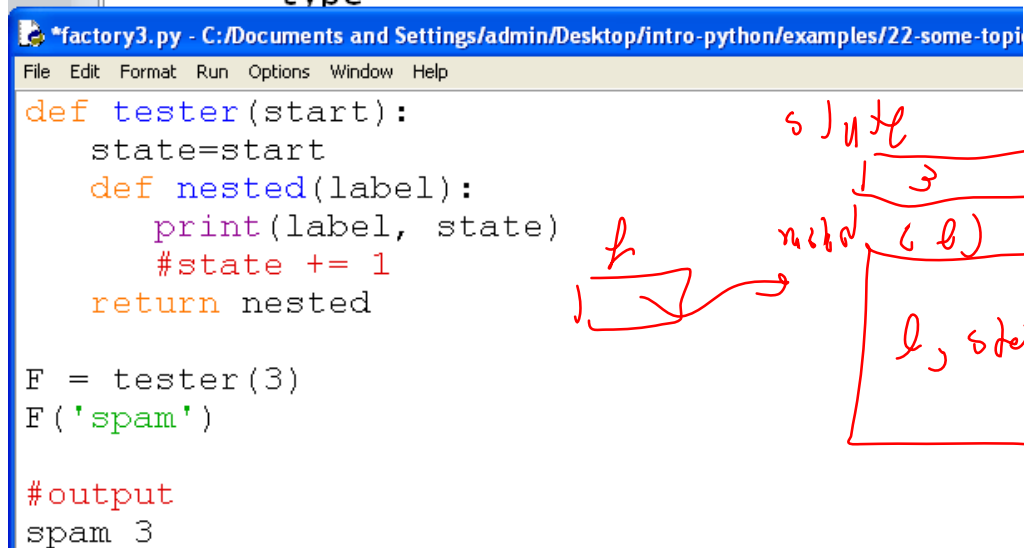
```
F = tester(0)  
F('spam') # F is a 'nested' with state attached  
F('ham')  
print(F.state) # Can access state outside functions too  
G = tester(42)  
G('eggs')  
print(G.state)  
F('ham')  
print(F.state)
```

```
#output  
spam 0  
ham 1  
2  
eggs 42  
43  
ham 2  
3
```



# Nonlocal variables

- Now see the another example.
- In this example, tester builds and returns the function nested, to be called later, and the state reference in nested maps the local scope of tester using the normal scope lookup rules



```
*factory3.py - C:/Documents and Settings/admin/Desktop/intro-python/examples/22-some-topi
File Edit Format Run Options Window Help
def tester(start):
    state=start
    def nested(label):
        print(label, state)
        #state += 1
    return nested

F = tester(3)
F('spam')
```

#output  
spam 3

Handwritten annotations: A box labeled 'f' points to the `nested` function definition. A box labeled 'state' points to the `state` variable in the `nested` function's local scope. A box labeled '3' points to the argument passed to `tester`.

## Nonlocal variables

- Changing a name in an enclosing def's scope is not allowed by default. Now, if we declare state in the tester scope as nonlocal within nested, we get to change it inside the nested function, too.
- If a variable is assigned in an enclosing def, it is nonlocal to nested functions.

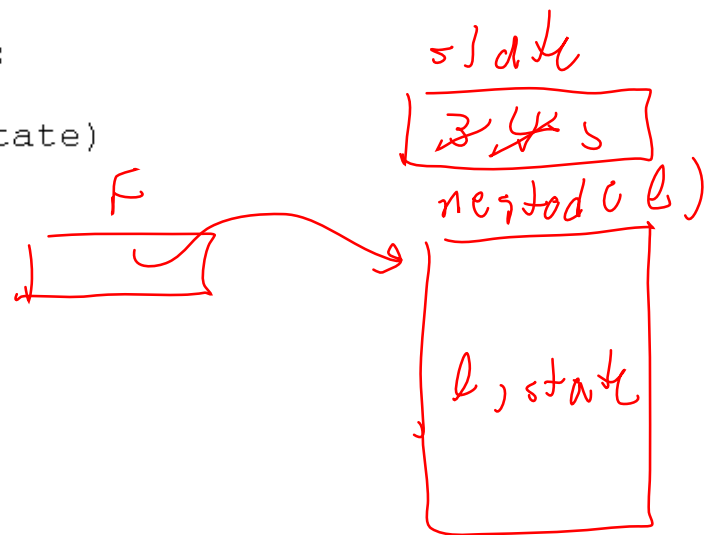
# Nonlocal variables

```
*factory4.py - C:/Documents and Settings/admin/Desktop/intro-python/examples/22-some-to
File Edit Format Run Options Window Help
def tester(start):
    state=start
    def nested(label):
        nonlocal state
        print(label, state)
        state += 1
    return nested

F = tester(3)
F('spam')
F('ham')
F('eggs')
```

#output

```
spam 3
ham 4
eggs 5
```



# Extended packing and Unpacking

- In Python, sequence assignment has been generalized to make this easier. In short, a single *starred name*, *\*X*, can be used in the assignment target in order to specify a more general matching against the sequence. The starred name is assigned a list, which collects all items in the sequence not assigned to other names. This is especially handy for common coding patterns such as splitting a sequence into its “front” and “rest,”.
- *\*\*X* can be used for dictionary unpacking

# Extended packing and Unpacking

```
seq = [1, 2, 3, 4]
a, b, c, d = seq
print(a, b, c, d)
#a, b = seq    #ValueError
a, *b = seq
print(a, b)
*a, b = seq
print(a, b)
a, *b, c = seq
print(a, b, c)
a, b, *c = seq
print(a, b, c)
```

**#output:**

```
1 2 3 4
1 [2, 3, 4]
[1, 2, 3] 4
1 [2, 3] 4
1 2 [3, 4]
```



# Extended packing Unpacking

- Naturally, like normal sequence assignment, extended sequence unpacking syntax works for any sequence types (really, again, any iterable), not just lists.

# Extended packing Unpacking

```
a, *b = 'spam'
print(a, b)
a, *b, c = 'spam'
print(a, b, c)
a, *b, c = range(4)
print(a, b, c)

L = [1, 2, 3, 4]
while L:
    front, *L = L
    print(front, L)

for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:
    print(a, b, c)
```

```
#output:
s ['p', 'a', 'm']
s ['p', 'a'] m
0 [1, 2] 3
```

```
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

```
1 [2, 3] 4
5 [6, 7] 8
```

# Extended packing Unpacking

```
seq = {1:'a', 2:'b', 3:'c', 4:'d', 5:'e'}  
a, b, c, *d = seq  
print(a, b, c, d)
```

```
1 2 3 [4, 5]
```

# Extended Sequence Unpacking

```
list1=[1,3,5]
a,b,c=[*list1]
print('unpacked: ', a,b,c)
a+=1; b+=1; c+=1
list2=[*list1, a, b ,c]
print('merged:   ', list2)
```

```
unpacked:  1 3 5
merged:    [1, 3, 5, 2, 4, 6]
```

# Extended Sequence Unpacking

```
d1 = {'a':1, 'b':2, 'c':3}
d2 = {'x':1, 'y':2, 'z':3}
a, b, c=**d1
print('unpacked: ', a, b, c)
d3=**d1
print('new dic: ', d3)
d4 = {**d1, **d2}
print('merge two dic: ', d4)
```

```
unpacked:  c a b
new dic:  {'c': 3, 'a': 1, 'b': 2}
merge two dic:  {'b': 2, 'y': 2, 'z': 3, 'a': 1, 'c': 3, 'x': 1}
```

# Arbitrary Arguments in functions

- The two matching extensions, `*` and `**`, are designed to support functions that take any number of arguments. Both can appear in either the function definition or a function call, and they have related purposes in the two locations.

## Arbitrary Arguments: packing

- The first use, in the function definition, collects unmatched positional arguments into a tuple:
- `>>> def f(*args): print(args)`
- When this function is called, Python collects all the positional arguments into a new tuple and assigns the variable `args` to that tuple. Because it is a normal tuple object, it can be indexed, stepped through with a for loop, and so on:

```
>>> f()
```

```
()
```

```
>>> f(1)
```

```
(1,)
```

```
>>> f(1, 2, 3, 4)
```

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

# Arbitrary Arguments: packing

```
def my sum(*args):  
    res = 0  
    for num in args:  
        res = res + num  
    return res
```

(1, 2)

```
s=my sum(1, 2)  
print('sum is: ', s)
```

(1, 2, 3, 4, 5)

```
s=my sum(1, 2, 3, 4, 5)  
print('sum is: ', s)
```

```
sum is: 3  
sum is: 15
```



# Arbitrary Arguments: unpacking

```
def fun(a, b, c, d):  
    print(a, b, c, d)
```

```
mylist = [1, 2, 3, 4]  
fun(*mylist)
```

1, 2, 3, 4

```
#output  
1 2 3 4
```

# Arbitrary Arguments

- The `**` feature is similar, but it only works for keyword arguments—it collects them into a new dictionary, which can then be processed with normal dictionary tools. In a sense, the `**` form allows you to convert from keywords to dictionaries, which you can then step through with `keys` calls, dictionary iterators, and the like (this is roughly what the `dict` call does when passed keywords, but it returns the new dictionary):

# Arbitrary Arguments: packing

```
>>> def f(**args): print(args)
```

```
>>> f()
```

```
{}
```

```
>>> f(a=1, b=2)
```

```
{'a': 1, 'b': 2}
```

*args = {'a': 1, 'b': 2}*

# Arbitrary Arguments: packing

```
# A Python program to demonstrate packing of
# dictionary items using **
def fun(**kwargs):

    # kwargs is a dict
    print(type(kwargs))

    # Printing dictionary items
    for key in kwargs:
        print("%s = %s" % (key, kwargs[key]))

# Driver code
fun(name="geeks", ID="101", language="Python")
```

```
<class 'dict'>
ID = 101
language = Python
name = geeks
```

kwargs: { 'name': 'geeks',  
          'ID': '101',  
          'language': 'python' }

# Arbitrary Arguments: unpacking

```
def fun(x, y, z):  
    print("x=" + str(x))  
    print("y=" + str(y))  
    print("z=" + str(z))  
  
myDict = {'x':1, 'y':2, 'z':3}  
fun(**myDict)
```

*'x':1 'y':2 'z':3*

```
x=1  
y=2  
z=3
```

# Arbitrary Arguments: packing

```
>>> def f(a, *pargs, **kargs): print(a, pargs, kargs)
```

```
>>> f(1, 2, 3, x=1, y=2)  
1 (2, 3) {'y': 2, 'x': 1}
```

*Handwritten diagram illustrating argument packing:*

<i>a</i>	<i>*pargs</i>	<i>**kargs</i>
<hr/>		
<i>↓</i>	<i>(2, 3)</i>	<i>{'x': 1, 'y': 2}</i>

```

def func(a, b, c, d): print(a, b, c, d)
args = (1, 2)
args += (3, 4)
func(*args) # Same as func(1, 2, 3, 4)
args = {'a': 1, 'b': 2, 'c': 3}
args['d'] = 4
func(**args)
func(*(1, 2), **{'d': 4, 'c': 3}) # Same as func(1, 2, d=4, c=3)
func(1, *(2, 3), **{'d': 4}) # Same as func(1, 2, 3, d=4)
func(1, c=3, *(2, ), **{'d': 4}) # Same as func(1, 2, c=3, d=4)
func(1, *(2, 3), d=4) # Same as func(1, 2, 3, d=4)
func(1, *(2, ), c=3, **{'d': 4}) # Same as func(1, 2, c=3, d=4)

```

```

def echo(*args, **kwargs): print(args, kwargs)
echo(1, 2, a=3, b=4)

```

```

def f(a, *b, c=6, **d): print(a, b, c, d)
f(1, 2, 3, x=4, y=5)
f(1, 2, 3, x=4, y=5, c=7)
f(1, 2, 3, c=7, x=4, y=5)

```

```

def f(a, c=6, *b, **d): print(a, b, c, d)
f(1, 2, 3, x=4)

```

```

def f(a, *b, c=6, **d): print(a, b, c, d)
f(1, *(2, 3), **dict(x=4, y=5))
f(1, *(2, 3), c=7, **dict(x=4, y=5))
f(1, *(2, 3), **dict(x=4, y=5, c=7))

```

$x < 4, y < 5, c < 7$

$(1, 2) \{ 'a': 3, 'b': 4 \}$

1  $(2, 3) 6 \{ 'x': 4, 'y': 5 \}$   
 1  $(2, 3) 7$  ✓

1  $(1, 3) 6 \{ 'x': 4 \}$

1  $(3, ) 2 \{ 'x': 4 \}$

✓  $(2, 3) 6 \{ 'x': 4, 'y': 5 \}$

✓  $(1, 3) 7$  ✓

✓  $(2, ) 7 \{ 'x': 4, 'y': 5, 'c': 7 \}$

# Arbitrary Arguments

```
#output
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
(1, 2) {'b': 4, 'a': 3}
1 (2, 3) 6 {'x': 4, 'y': 5}
1 (2, 3) 7 {'x': 4, 'y': 5}
1 (2, 3) 7 {'x': 4, 'y': 5}
1 (3,) 2 {'x': 4}
1 (2, 3) 6 {'x': 4, 'y': 5}
1 (2, 3) 7 {'x': 4, 'y': 5}
1 (2, 3) 7 {'x': 4, 'y': 5}
```



# Function Annotations

- It's also possible to attach *annotation information*—*arbitrary* user-defined data about a function's arguments and result—to a function.

```
def func(a: 'spam', b: (1, 10), c: float) -> int:
    return a + b + c
print(func(1, 2, 3))

#output
6
```

# isinstance

- Returns a Boolean stating whether the object is an instance or subclass of another object.
- Syntax:  
`isinstance (object, classinfo)`

# isinstance

```
i=22
print(isinstance(i,int))
print(isinstance(1,type(55)))
print(isinstance(1, (int, float)))
print(isinstance('Ni', (int, float)))
print(isinstance(42, str))
print(isinstance('x', str))
print(isinstance(b'x', str))
```

```
#output
True
True
True
False
False
True
False
```

# isinstance

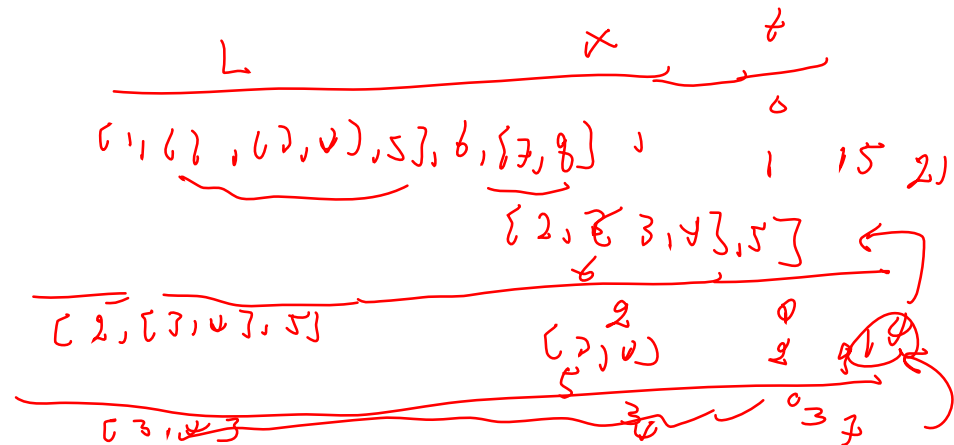
```
def sumtree(L):
    tot = 0
    for x in L:
        if not isinstance(x, list):
            tot += x
        else:
            tot += sumtree(x)
    return tot
L = [1, [2, [3, 4], 5], 6, [7, 8]]
print(sumtree(L))

# Pathological cases
print(sumtree([1, [2, [3, [4, [5]]]]]))
print(sumtree([[[[1], 2], 3], 4], 5]))
```

# For each item at this level  
# Add numbers directly  
# Recur for sublists  
# Arbitrary nesting  
# Prints 36  
# Prints 15 (right-heavy)  
# Prints 15 (left-heavy)

#output

36  
15  
15



# Iterators

- In the preceding chapter, it is mentioned that the for loop can work on any sequence type in Python, including lists, tuples, and strings, like this:

```
for x in [1, 2, 3, 4]: print(x ** 2, end=' ')
```

1 4 9 16

- Actually, the for loop turns out to be even more generic than this—it works on any *iterable object*. *In fact, this is true of all iteration tools that scan objects from left to right* in Python, including for loops, the list comprehensions, the map built-in function, and more.

# Iterators

- An iterator is an object that allows a programmer to traverse through all the elements of a collection regardless of its specific implementation.
- Technically speaking, Python iterator object must implement two special methods, `__iter__()` and `__next__()`, collectively called the iterator protocol.
- An object is called iterable if we can get an iterator from it. Most of built-in containers in Python like: list, tuple, string etc. are iterables. The `iter()` function (which in turn calls the `__iter__()` method) returns an iterator from them.

# Iterators

```
L = [1, 2, 3]
I = __iter__(L)      # Obtain an iterator object from an iterable
print(I.__next__())  # Call iterator's next to advance to next item
print(I.__next__())
print(I.__next__())
```

#output

```
1
2
3
```

```
L = [1, 2, 3]
I = iter(L)          # Obtain an iterator object from an iterable
print(next(I))        # Call iterator's next to advance to next item
print(next(I))
print(next(I))
```

#output

```
1
2
3
```

## Other Built-in Type Iterables

- Besides physical sequences like lists, other types have useful iterators as well.
- The classic way to step through the keys of a *dictionary*, for example, is to request its keys list explicitly.



```
D = {'a':1, 'b':2, 'c':3}
for key in D.keys():
    print(key, D[key])
I = iter(D)
print(next(I))
print(next(I))
print(next(I))

R = range(5)
print(R)

I = iter(R)
print(next(I))
print(next(I))
print(list(range(5)))
```

**#output**

```
a 1
b 2
c 3
a
b
c
range(0, 5)
0
1
[0, 1, 2, 3, 4]
```

## Other Built-in Type Iterables

```
E = enumerate('spam') # enumerate is an iterable too
print(E)
I = iter(E)
print(next(I))
print(next(I))
print(list(enumerate('spam')))
```

*for x in enumerate('spam')*

*(0, 's')*

*(1, 'p')*

```
#output
<enumerate object at 0x017BFFD0>
(0, 's')
(1, 'p')
[(0, 's'), (1, 'p'), (2, 'a'), (3, 'm')]
```

```

Z = zip((1, 2, 3), (10, 20, 30)) # zip is the same: a one-pass iterator
print(list(Z))
print('1---')
Z = zip((1, 2, 3), (10, 20, 30)) # Manual iteration (iter() not needed)
for pair in Z: print(pair)      # Exhausted after one pass
print('2---')
Z = zip((1, 2, 3), (10, 20, 30)) # Manual iteration (iter() not needed)
print(next(Z))
print(next(Z))
print('3---')
Z = zip((1, 2, 3), (10, 11, 12))
I1 = iter(Z)
I2 = iter(Z)                      # Two iterators on one zip
print(next(I1))
print(next(I1))
print(next(I2))
print('4---')
M = map(abs, (-1, 0, 1))          # Ditto for map (and filter)
I1 = iter(M); I2 = iter(M)
print(next(I1), next(I1), next(I1))
print('5---')
R = range(3)                      # But range allows many iterators
I1, I2 = iter(R), iter(R)
print([next(I1), next(I1), next(I1)])
print(next(I2))                  # Multiple active scans, like 2.X lists

```

## Other Built-in Type Iterables

```
#output
[(1, 10), (2, 20), (3, 30)]
1---
(1, 10)
(2, 20)
(3, 30)
2---
(1, 10)
(2, 20)
3---
(1, 10)
(2, 11)
(3, 12)
4---
1 0 1
5---
[0, 1, 2]
0
```