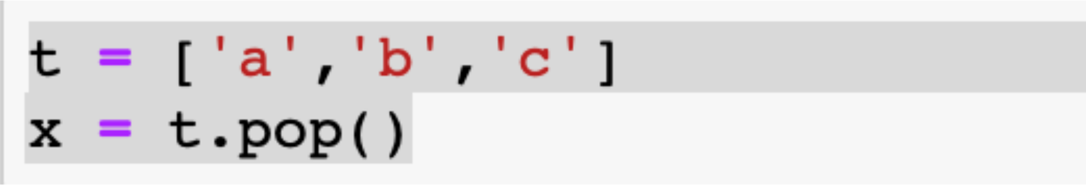
# MSML 605
# Python Contd.

- Create a list t, 1,5,6,7
- Print t
- copy t to r list
- print r
- Modify second element of r
- print r
- print t

- What do you notice?
- r = t[:]

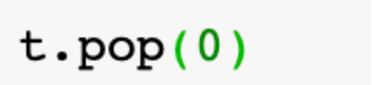
# Deleting Elements

- Pop



```
t = ['a','b','c']
x = t.pop()
```

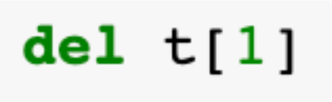
- Pop modifies the list and returns the element that was removed.



```
t.pop(0)
```

removes the second element

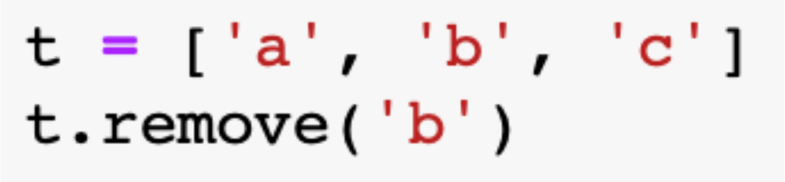
- del also deletes elements, when you don’t need them



```
del t[1]
```

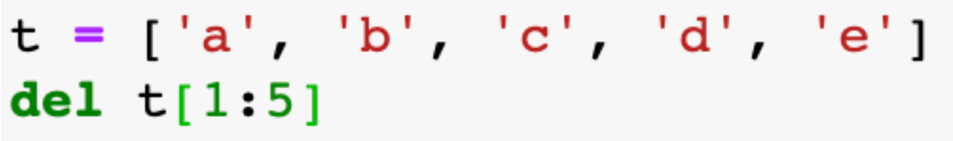
# Remove

- If you know the element you want to remove (but not the index), use remove:



```
t = ['a', 'b', 'c']
t.remove('b')
```

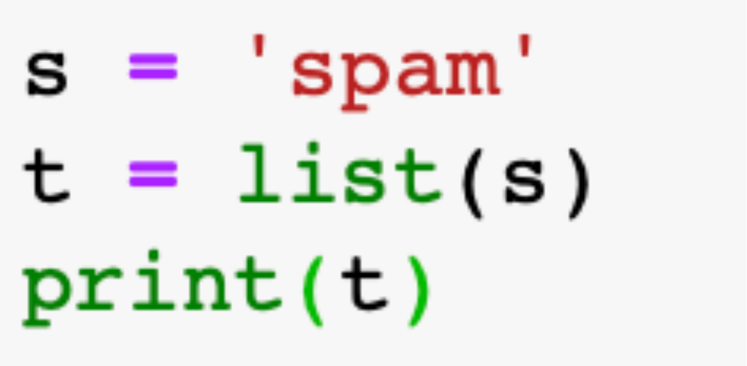
- The return value from remove is None
- To remove more than one element, use del



```
t = ['a', 'b', 'c', 'd', 'e']
del t[1:5]
```

# Strings and Lists

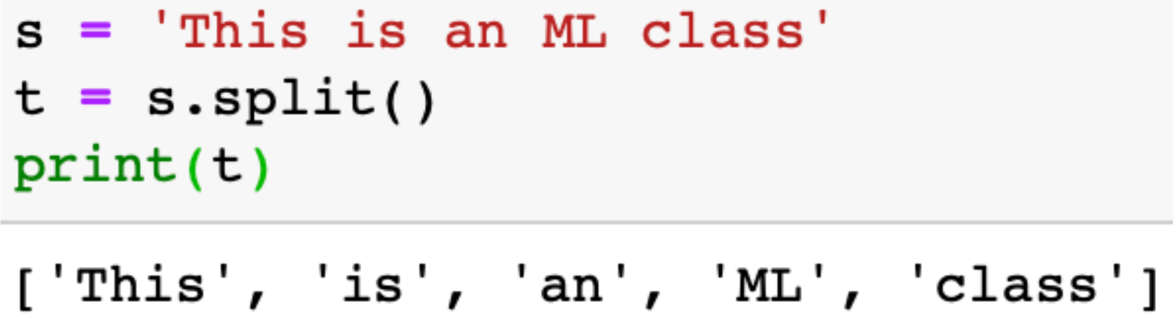
- A string is a sequence of characters
- A list is a sequence of values
- A list of characters is not the same as a string.



```
s = 'spam'
t = list(s)
print(t)
```

# Split Method

- split method

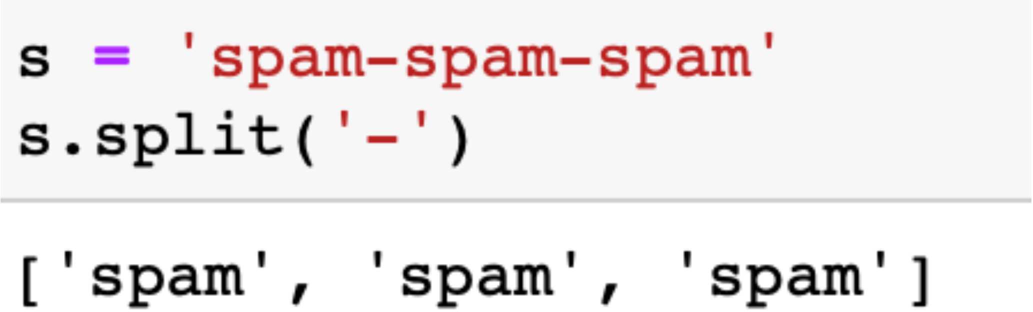


```python
s = 'This is an ML class'
t = s.split()
print(t)
```

```
['This', 'is', 'an', 'ML', 'class']
```

# Delimiter

- A delimiter specifies which characters to use as word boundaries



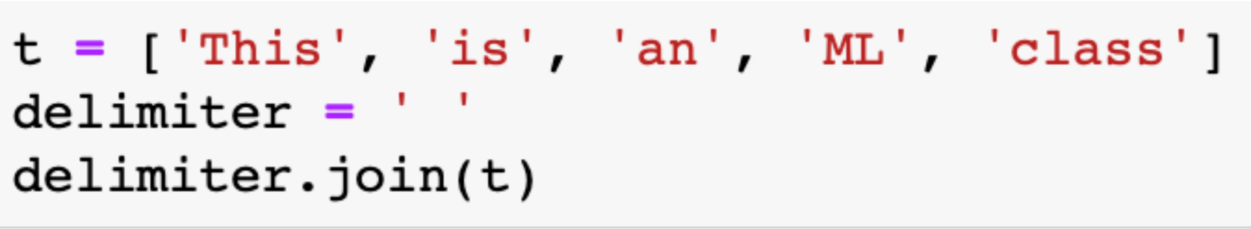
```
s = 'spam-spam-spam'
s.split('-')
```

```
['spam', 'spam', 'spam']
```

# Join

- join is the inverse of split.

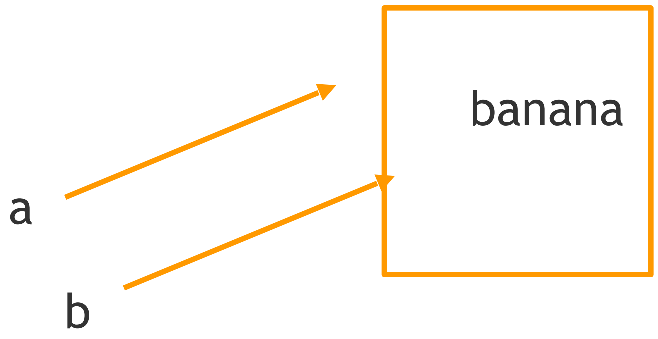
- It takes a list of strings and concatenates the elements.



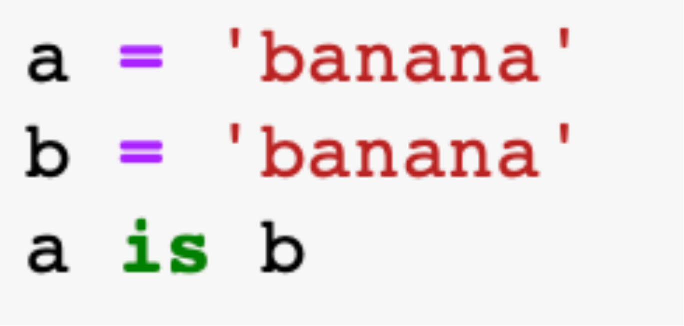
```
t = ['This', 'is', 'an', 'ML', 'class']
delimiter = ' '
delimiter.join(t)
```

# Objects and values

- a = ‘banana’
  b = ‘banana’



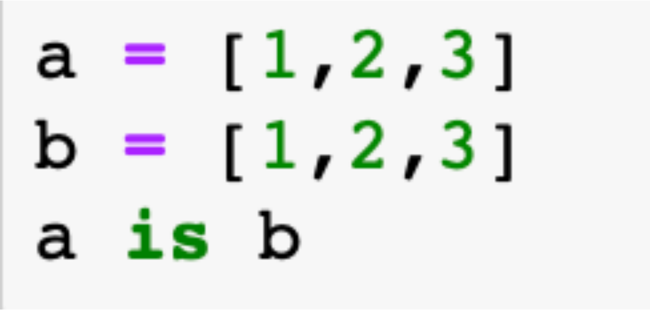
- a and b both refer to a string, but we don’t know whether they refer to the same string
- To determine, we can use, ‘is’ operator



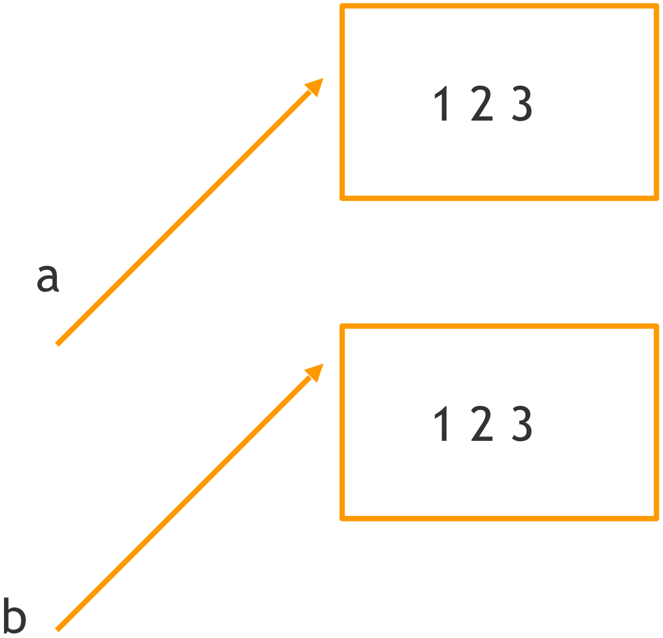
```
a = 'banana'
b = 'banana'
a is b
```

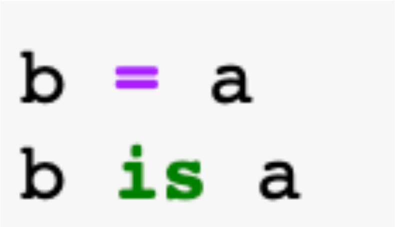
# Objects and values

- when you create two lists, you get two objects:

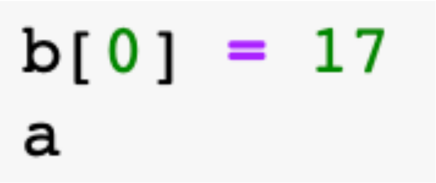


```
a = [1,2,3]
b = [1,2,3]
a is b
```





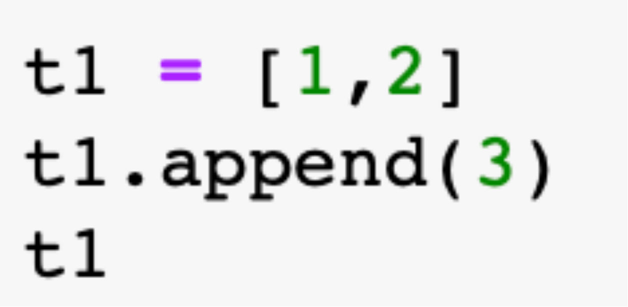
```
b = a
b is a
```



```
b[0] = 17
a
```

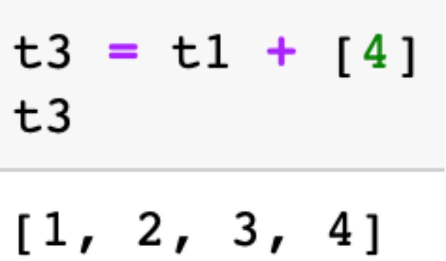
# List Arguments

- When you pass a list to a function, the function gets a reference to the list.
- If the function modifies a list parameter, the caller sees the change.
- Some operations modify lists and other operations create new lists.
- append method modifies a list, but the + operator creates a new list:



```
t1 = [1,2]
t1.append(3)
t1
```

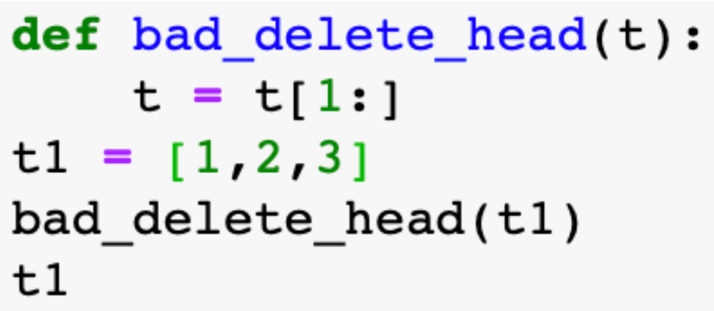
# List Arguments



```
t3 = t1 + [4]
t3
```

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

- The difference is important when you write functions that are supposed to modify lists.



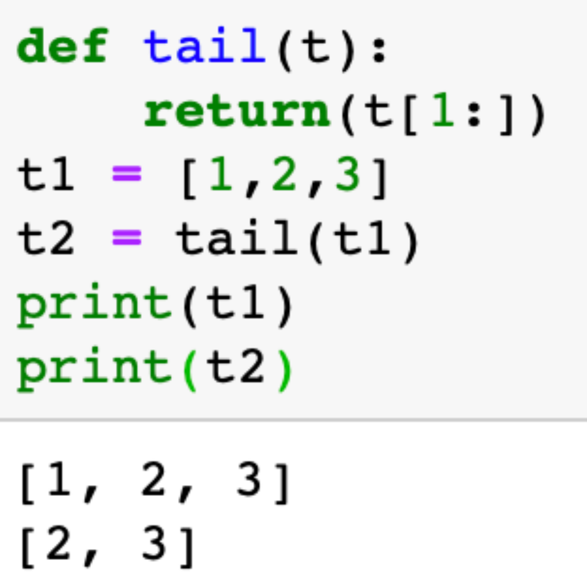
```
def bad_delete_head(t):
    t = t[1:]
t1 = [1,2,3]
bad_delete_head(t1)
t1
```

  The slice operator creates a new list and the assignment makes t refer to it.

- None of that has any effect on the list passed as an argument.

# List Arguments

- if we want to slice a list we can return it



```
def tail(t):
    return(t[1:])
t1 = [1,2,3]
t2 = tail(t1)
print(t1)
print(t2)
```

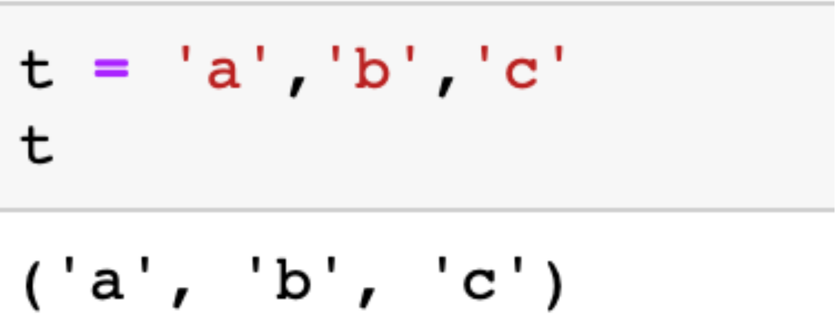
```
[1, 2, 3]
[2, 3]
```

- The list leaves the original list unmodified

# Tuples

# Introduction

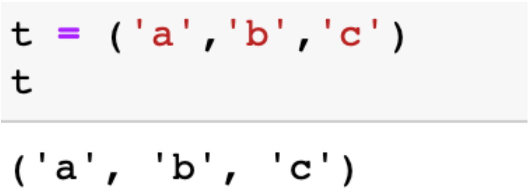
- A tuple is a sequence of values
- They are indexed and a lot like lists
- A comma-separated list of values



```
t = 'a','b','c'
t
```

```
('a', 'b', 'c')
```

- It is common to enclose tuples in parentheses:

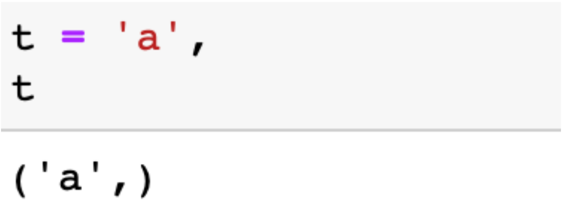


```
t = ('a','b','c')
t
```

```
('a', 'b', 'c')
```

# Tuples

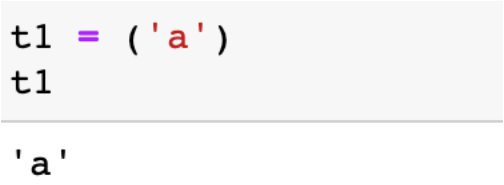
- To create a tuple with a single element, you have to include a final comma



```
t = 'a',
t
```

```
('a',)
```

- A single value in parentheses is not a tuple:

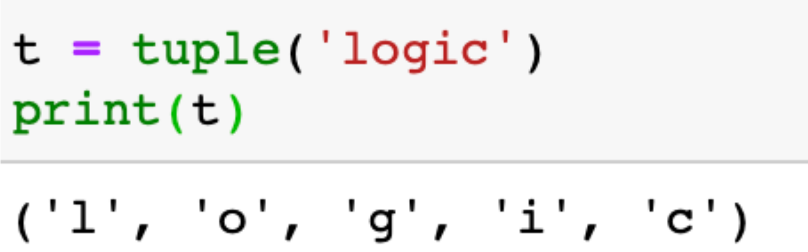


```
t1 = ('a')
t1
```

```
'a'
```

# Tuples - Index Operator

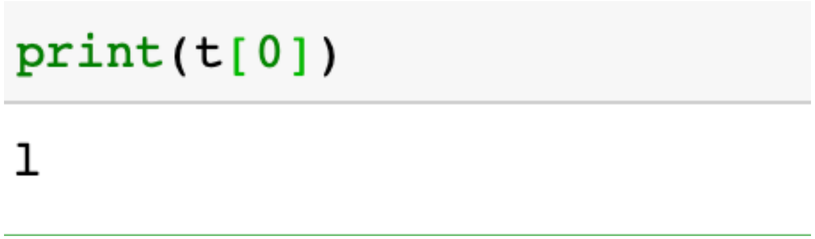
- If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence



```
t = tuple('logic')
print(t)
```

```
('l', 'o', 'g', 'i', 'c')
```

- Most list operators also work on tuples

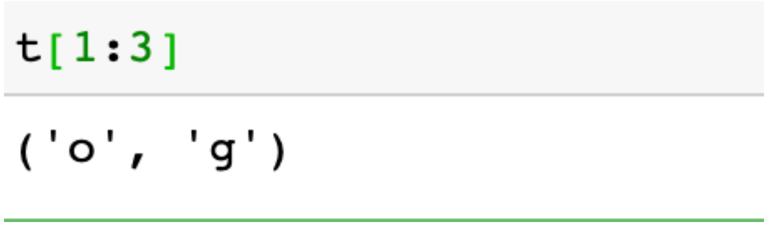


```
print(t[0])
```

```
l
```

# Tuples - Slice Operator

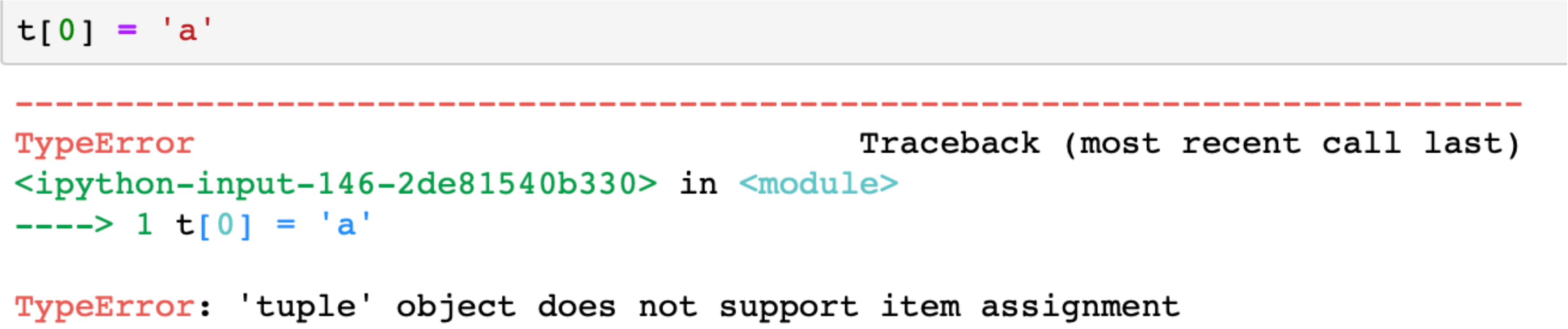
- Slicing



```
t[1:3]
```

```
('o', 'g')
```

- If you try to modify one of the elements of the tuple:



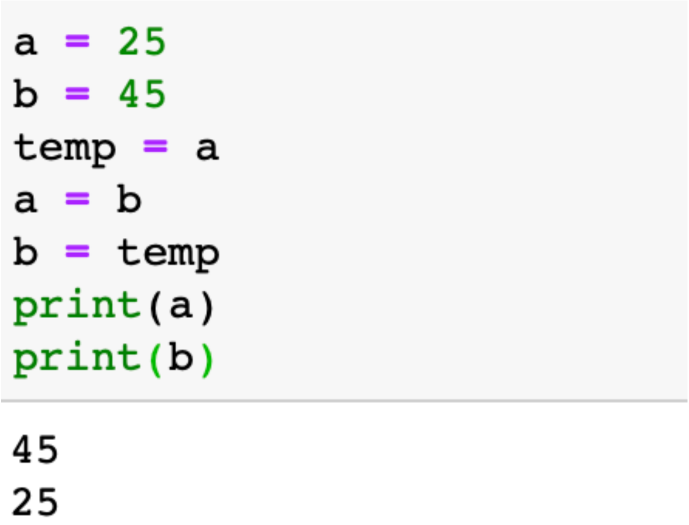
```
t[0] = 'a'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-146-2de81540b330> in <module>
----> 1 t[0] = 'a'

TypeError: 'tuple' object does not support item assignment
```

- Tuples are immutable

# Tuple - Assignment

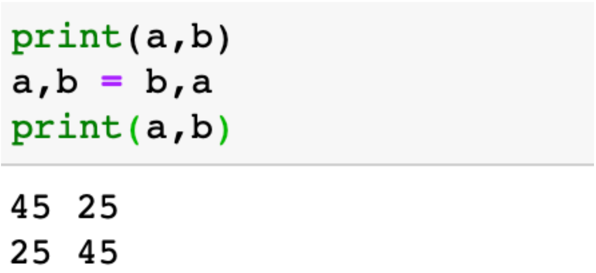
- If we want to swap two variables we will need a third variable, for example



```
a = 25
b = 45
temp = a
a = b
b = temp
print(a)
print(b)
```

```
45
25
```

- With tuples it is more elegant

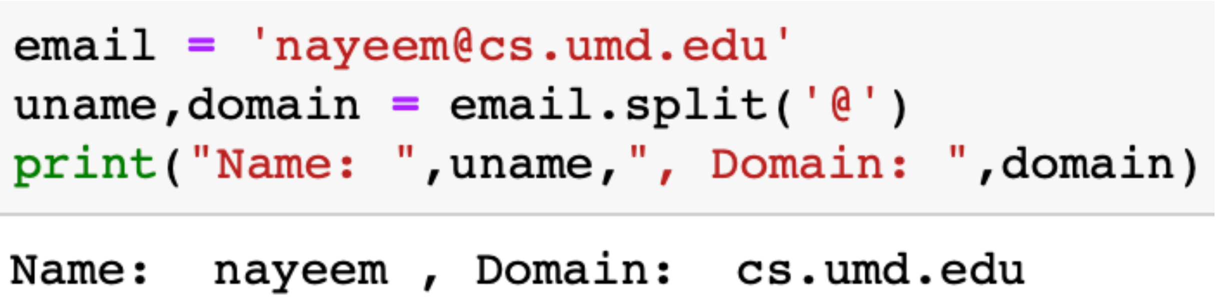


```
print(a,b)
a,b = b,a
print(a,b)
```

```
45 25
25 45
```

# Tuple - Assignment

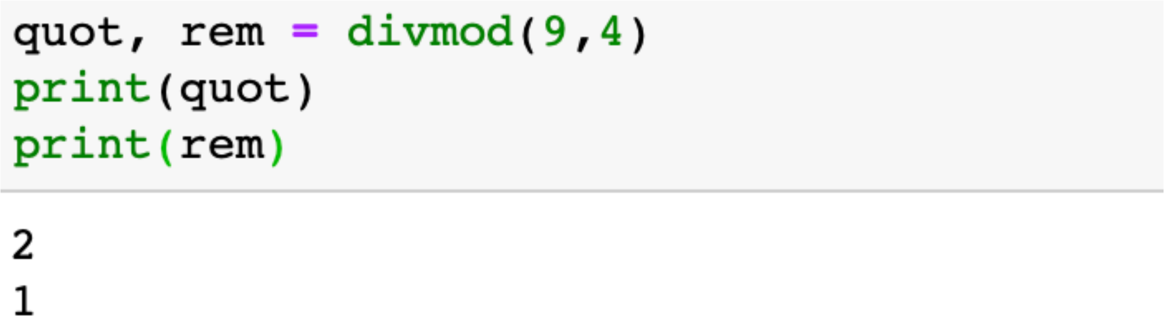
- The right side can be any kind of sequence (string, list, or tuple)



```
email = 'nayeem@cs.umd.edu'
uname,domain = email.split('@')
print("Name: ",uname,", Domain: ",domain)
```

```
Name:  nayeem , Domain:  cs.umd.edu
```

# Tuples as Return Values

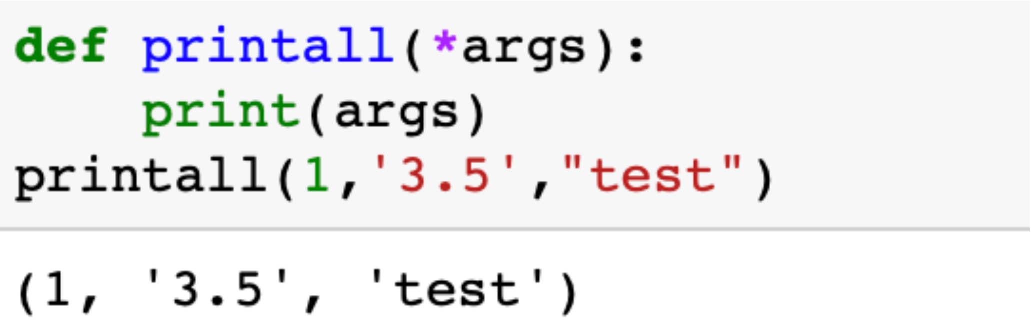


```
quot, rem = divmod(9,4)
print(quot)
print(rem)
```

```
2
1
```

# Variable-length argument tuples

- Functions can take a variable number of arguments.
- A parameter name that begins with a * gathers arguments into a tuple, for example

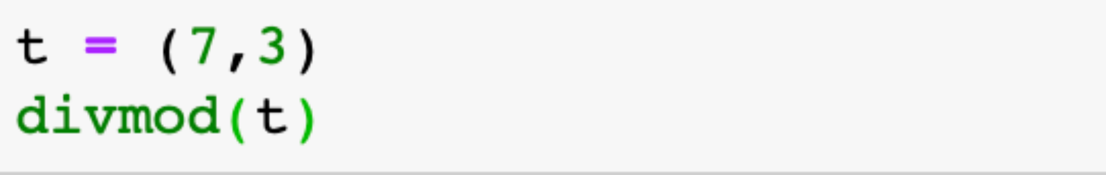


```
def printall(*args):
    print(args)
printall(1,'3.5',"test")
```

```
(1, '3.5', 'test')
```

# Scatter

- The complement of gather is scatter.
- If you have a sequence of values and you want to pass it to a function as multiple arguments, use * operator

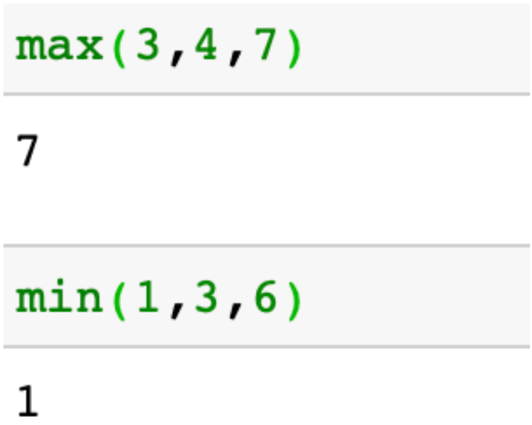


```
t = (7,3)
divmod(t)
```

- What do you notice?

# Variable length arguments

- Many of the built-in functions use variable-length argument tuples.
- for example, max and min can take any number of arguments:



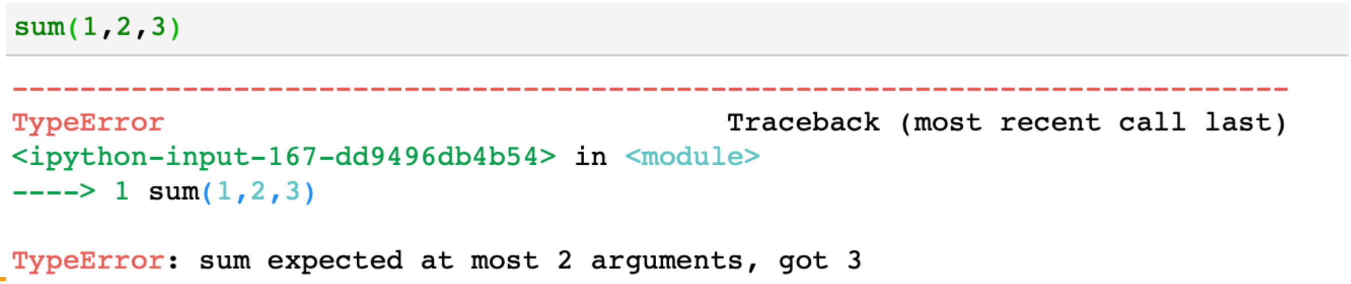
```
max(3,4,7)
```

```
7
```

```
min(1,3,6)
```

```
1
```

- sum cannot



```
sum(1,2,3)
```

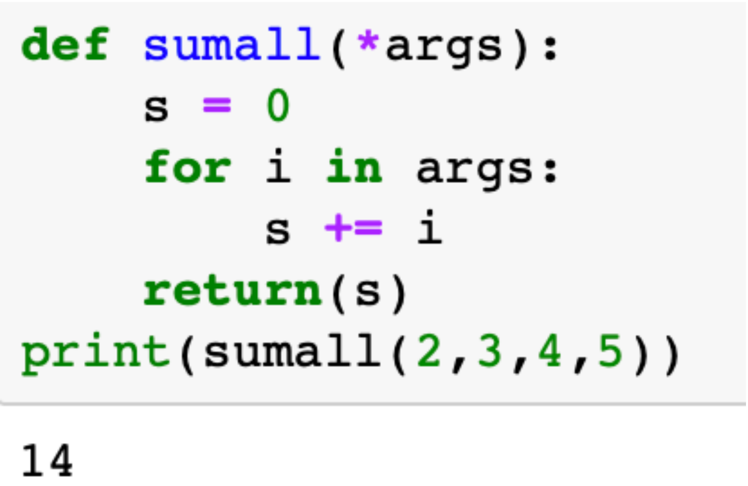
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-167-dd9496db4b54> in <module>
----> 1 sum(1,2,3)

TypeError: sum expected at most 2 arguments, got 3
```

# Variable length Tuples

- Write a function called sumall that takes any number of arguments and returns their sum.

# Variable length Tuples

- Write a function called sumall that takes any number of arguments and returns their sum.



```
def sumall(*args):
    s = 0
    for i in args:
        s += i
    return(s)
print(sumall(2,3,4,5))
```

```
14
```

# Lambda Functions

- Lambda is a way to create small anonymous functions

- They are created where they are needed.

- Lambda functions are used in combination with the functions filter(), map(), and reduce().

# Lambda Functions

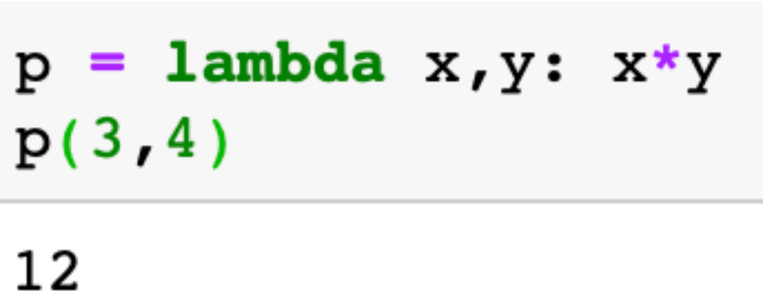
- Syntax:

  lambda ‹argument list›: ‹expression›

- argument list consists of a comma separated list of arguments

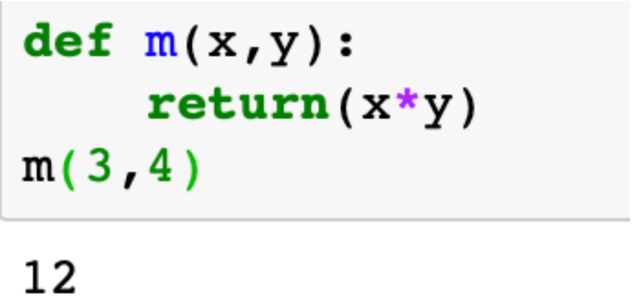
- Expression is an arithmetic expression using these arguments.

# Example



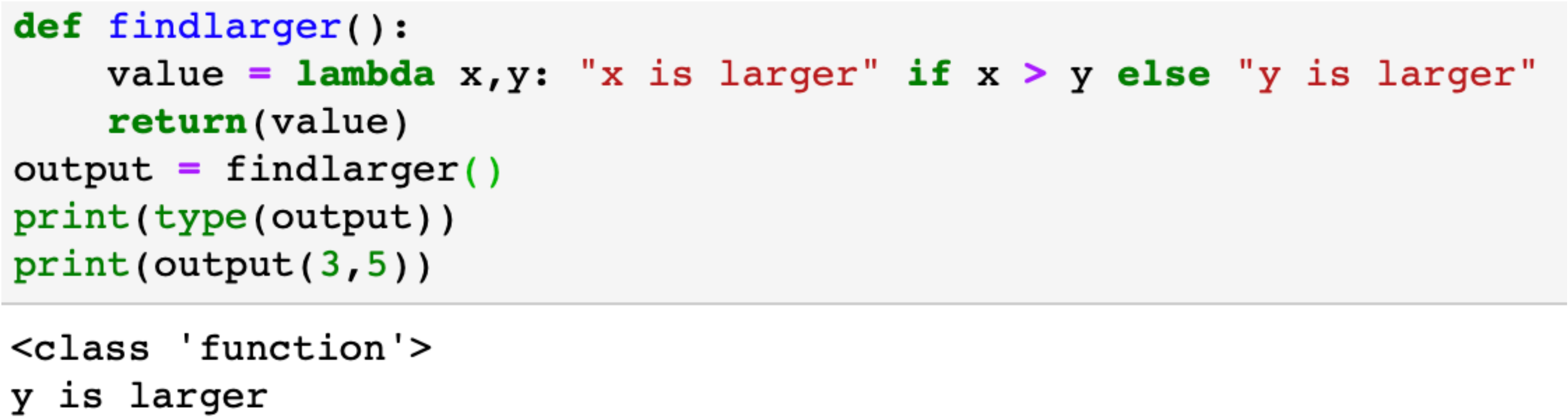
```python
p = lambda x,y: x*y
p(3,4)
```

```
12
```



```python
def m(x,y):
    return(x*y)
m(3,4)
```

```
12
```



```python
def findlarger():
    value = lambda x,y: "x is larger" if x > y else "y is larger"
    return(value)
output = findlarger()
print(type(output))
print(output(3,5))
```

```
<class 'function'>
y is larger
```

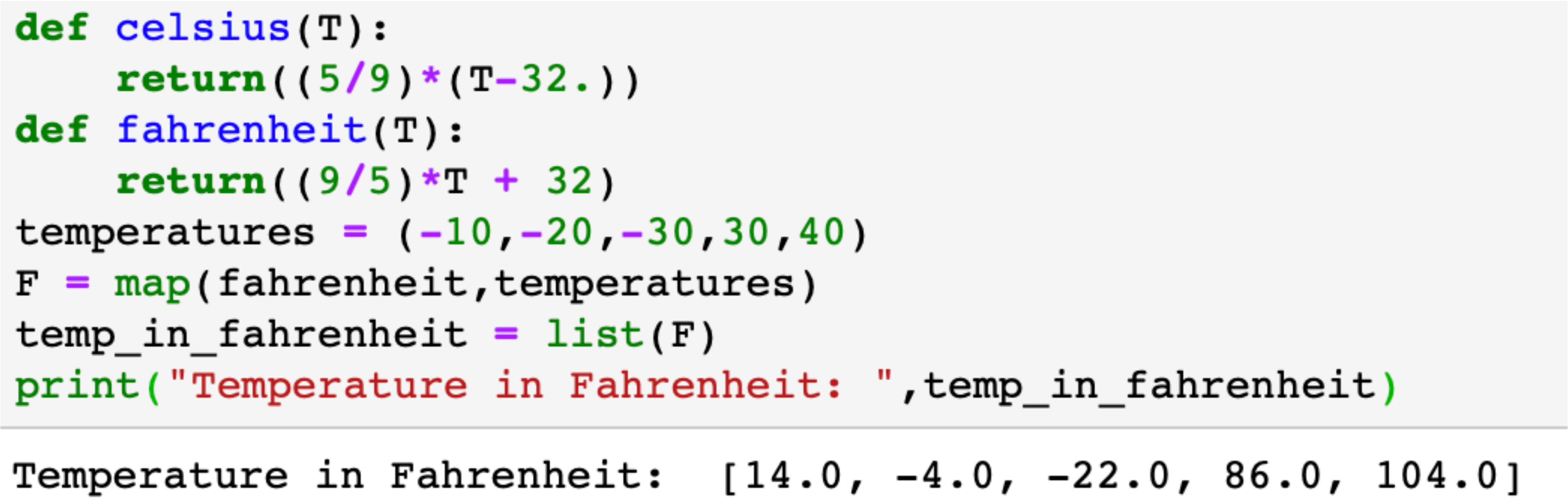
# Example

- Advantage of lambda can be seen when it is used in combination with map
- map() is a function with two arguments

```
r = map(func, seq)
```

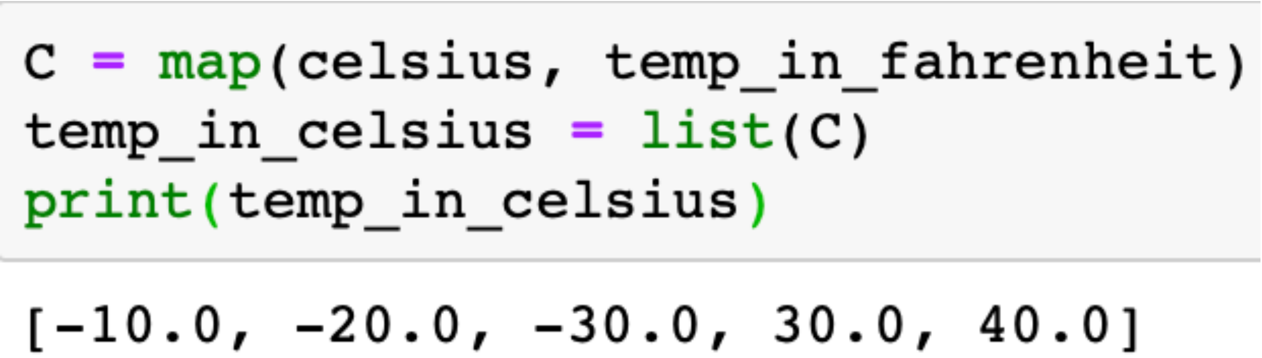
– the first argument func is the name of a function
– and the second a sequence (e.g., a list) seq.

# map Functions



```python
def celsius(T):
    return((5/9)*(T-32.))
def fahrenheit(T):
    return((9/5)*T + 32)
temperatures = (-10,-20,-30,30,40)
F = map(fahrenheit,temperatures)
temp_in_fahrenheit = list(F)
print("Temperature in Fahrenheit: ",temp_in_fahrenheit)
```

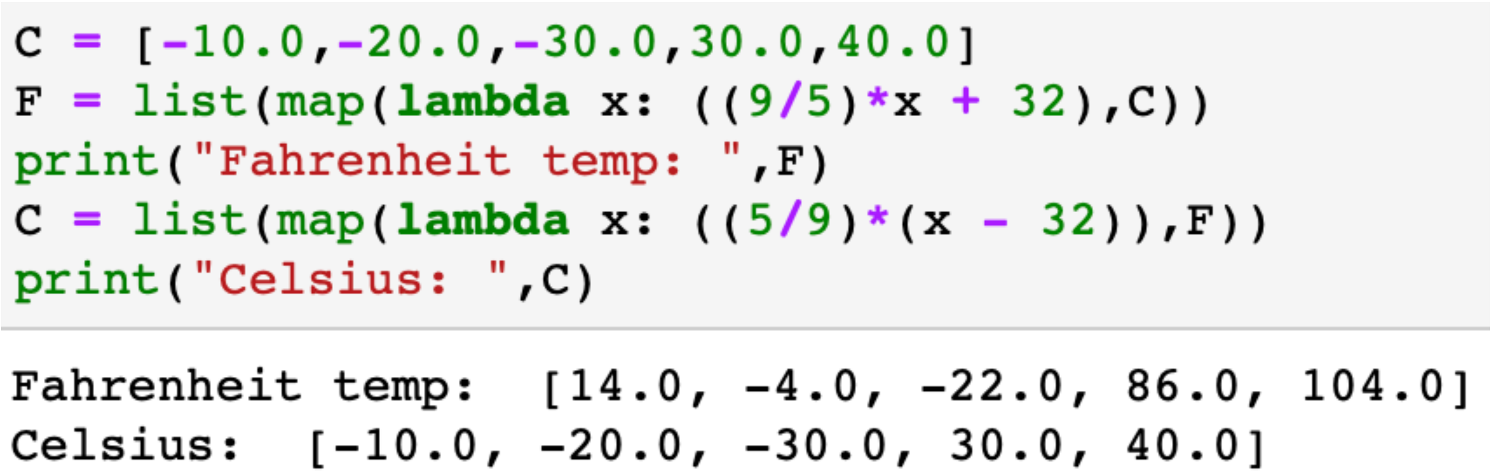
```
Temperature in Fahrenheit:  [14.0, -4.0, -22.0, 86.0, 104.0]
```



```python
C = map(celsius, temp_in_fahrenheit)
temp_in_celsius = list(C)
print(temp_in_celsius)
```

```
[-10.0, -20.0, -30.0, 30.0, 40.0]
```

# lambda with map



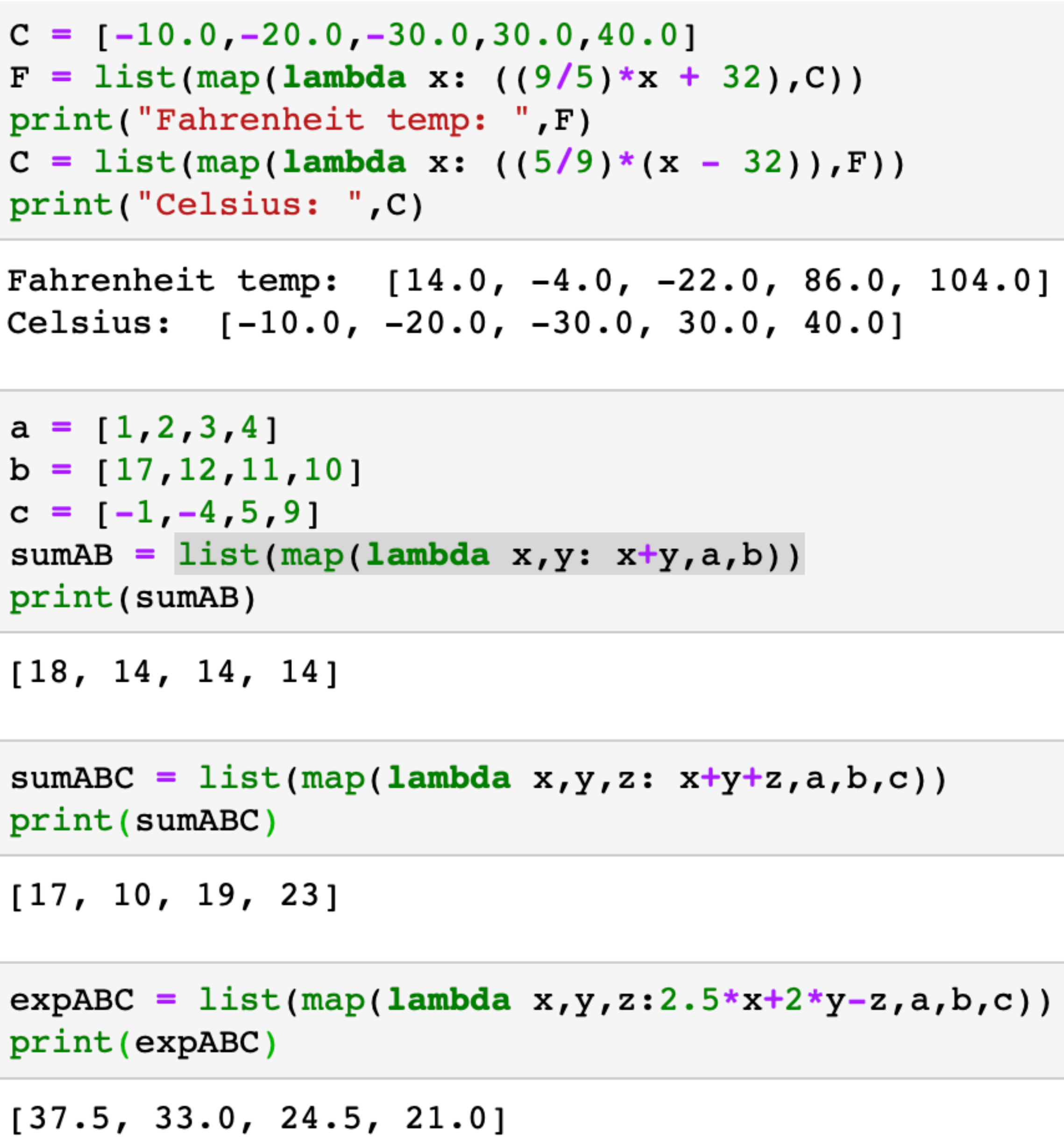
```
C = [-10.0,-20.0,-30.0,30.0,40.0]
F = list(map(lambda x: ((9/5)*x + 32),C))
print("Fahrenheit temp: ",F)
C = list(map(lambda x: ((5/9)*(x - 32)),F))
print("Celsius: ",C)
```

```
Fahrenheit temp:  [14.0, -4.0, -22.0, 86.0, 104.0]
Celsius:  [-10.0, -20.0, -30.0, 30.0, 40.0]
```

# Map

- map() can be applied to more than one list.
- The lists have to have the same length.
- map() will apply its lambda function to the elements of the argument lists
- It first applies to the elements of the 0th index, then to the elements with the 1st index, so on

# Maps List



```
C = [-10.0,-20.0,-30.0,30.0,40.0]
F = list(map(lambda x: ((9/5)*x + 32),C))
print("Fahrenheit temp: ",F)
C = list(map(lambda x: ((5/9)*(x - 32)),F))
print("Celsius: ",C)
```

```
Fahrenheit temp:  [14.0, -4.0, -22.0, 86.0, 104.0]
Celsius:  [-10.0, -20.0, -30.0, 30.0, 40.0]
```

```
a = [1,2,3,4]
b = [17,12,11,10]
c = [-1,-4,5,9]
sumAB = list(map(lambda x,y: x+y,a,b))
print(sumAB)
```

```
[18, 14, 14, 14]
```

```
sumABC = list(map(lambda x,y,z: x+y+z,a,b,c))
print(sumABC)
```

```
[17, 10, 19, 23]
```

```
expABC = list(map(lambda x,y,z:2.5*x+2*y-z,a,b,c))
print(expABC)
```

```
[37.5, 33.0, 24.5, 21.0]
```

# Filtering

- filter function filters out all the elements of a list, for which function returns True.

  ```
  filter(<function>, list)
  ```

- function, f, is the first argument.
- f returns a Boolean value, i.e. either True or False

- This function will be applied to every element of the list.

- Only if f returns True will the element of the list be included in the result list.

# Filtering

```python
data = [1,3,4,8,5,26]

odd_numbers = list(filter(lambda x : x%2, data))

even_numbers = list(filter(lambda x: x%2==0, data))

print(odd_numbers)

print(even_numbers)
```

# Reduce

- Function reduce, continually applies function to the sequence
  reduce（func, seq）
- if seq＝[s1,s2,s3, … ,sn], calling reduce（func, seq）works like this：
  - at first, func will be applied to s1 and s2
  - next step, func will be applied to result of step 1 result and s3, so on

# Reduce

```
from functools import reduce

m = reduce(lambda x,y:x+y,[34,43,56,76])
print(m)

sum = reduce(lambda x,y: x+y , range(1,101))
print(sum)

largest = reduce(lambda x,y : x if x > y else y, [3,25,23,12,4,9])
print(largest)
```

# Array

- import array as array

- array(data type, list)

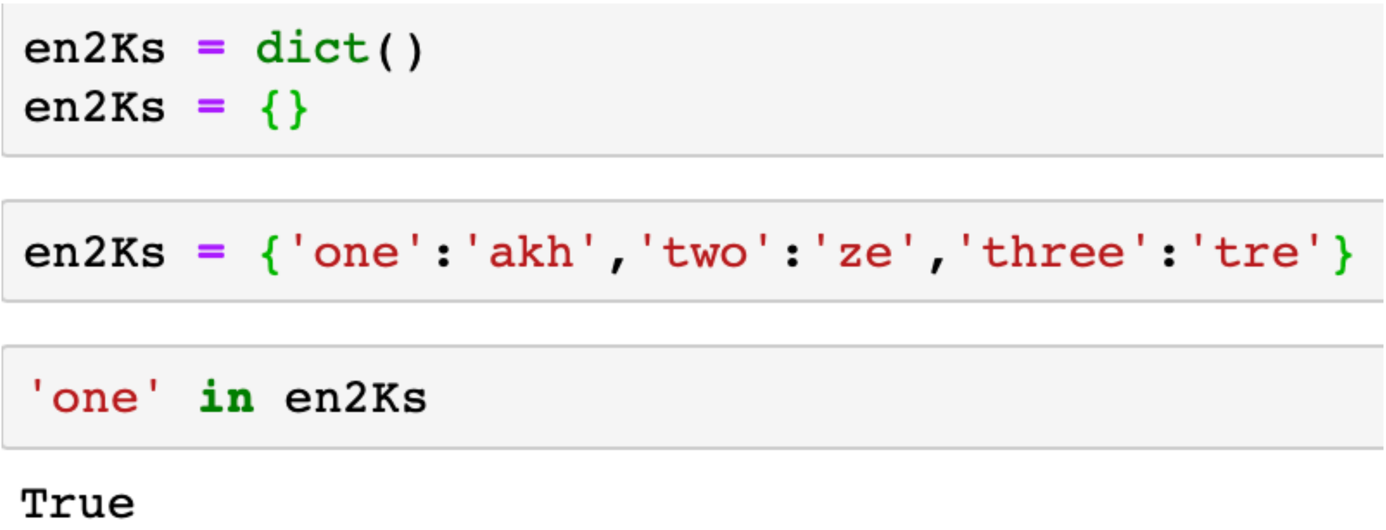
```
a = array('f',[2,4,6,8])

array('f', [2.0, 4.0, 6.0, 8.0])

help(array)
```

# Dictionaries

# Introduction

- A dictionary is like a list.
- In a list, the indices have to be integers.
- In a dictionary they can be almost any type.
- This set of indices are called keys.
- And dictionary is a mapping between keys and values
- Each key maps to a value.

# Initialization



```
en2Ks = dict()
en2Ks = {}
```

```
en2Ks = {'one':'akh','two':'ze','three':'tre'}
```

```
'one' in en2Ks
```

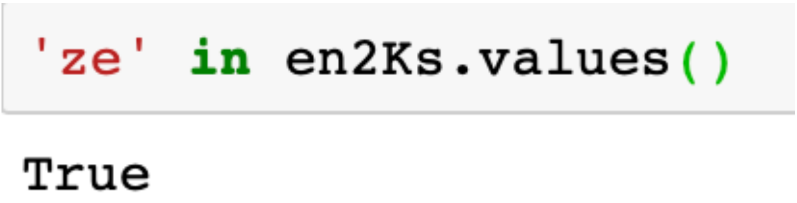
```
True
```

- The ‘in’ operator works on the keys in a dictionary

  ‘one’ in en2Ks

# Values

- To see whether a value exists, use a method called values



```
'ze' in en2Ks.values()
```

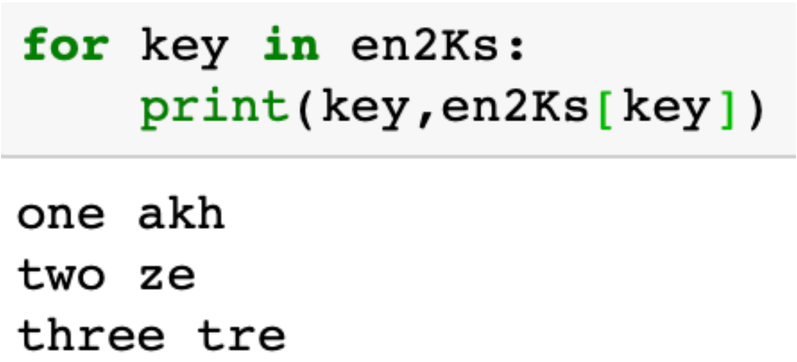
```
True
```

# ‘in’ operator algorithms

- ‘in’ operator uses different algorithms for lists and dictionaries.
- For lists, it uses a search algorithm
- For dictionaries Python uses a hashtable
- In a hashtable, the ‘in’ operator takes about the same time no matter how many items there are in a dictionary.

# Looping and Dictionaries

- You can use a ‘for’ loop to traverse the keys of a dictionary



```
for key in en2Ks:
    print(key,en2Ks[key])
```

```
one akh
two ze
three tre
```

- Dictionaries have a method called keys that returns the keys of the dictionary, in no particular order, as a list

# Reverse LookUp

- Given a dictionary ‘d’ and a key ‘k’
- We can find the value using
  v = d[k]
  This is called lookup
- If you have v and you want to find k, you have two problems:
  - there might be more than one key that maps to the value v
  - there is no simple syntax for reverse lookup, you have to search for it.

# Dictionaries and Lists

- Lists can appear as values in a dictionary
- Consider a dictionary that maps frequencies to letters
- A frequency may be mapped to several letters.
- In order to represent such a mapping, the values (letters) should be a list of letters.

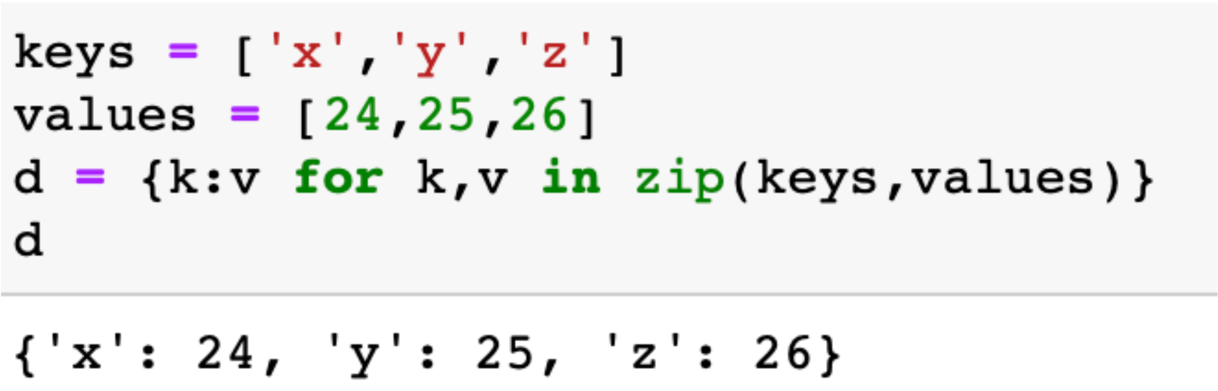
# Dictionaries and Lists

- Can lists be keys?

  ```
  t = [1,2,3]
  d = dict()
  d[t] = ‘oops’
  ```

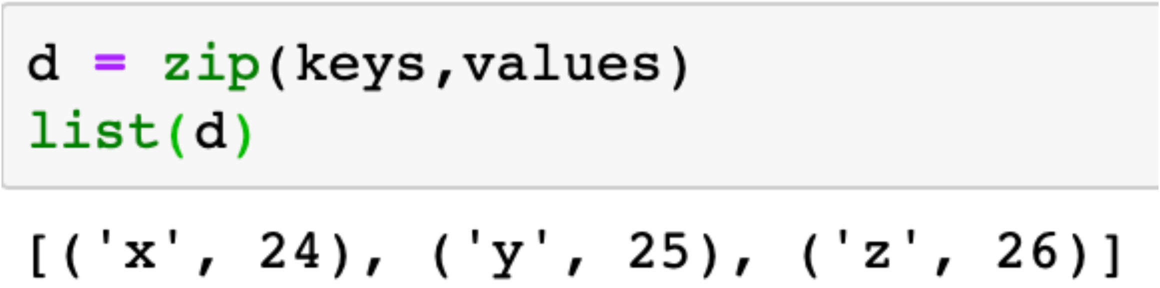
  What do you expect?

# Hashing from two arrays



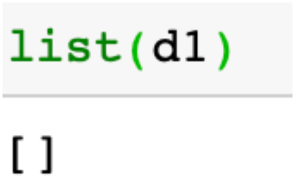
```
keys = ['x','y','z']
values = [24,25,26]
d = {k:v for k,v in zip(keys,values)}
d
```

```
{'x': 24, 'y': 25, 'z': 26}
```



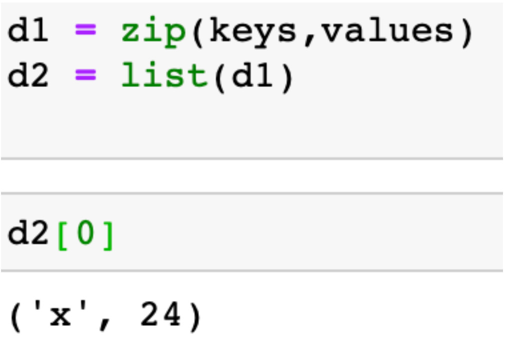
```
d = zip(keys,values)
list(d)
```

```
[('x', 24), ('y', 25), ('z', 26)]
```



```
list(d1)
```

```
[]
```



```
d1 = zip(keys,values)
d2 = list(d1)
```

```
d2[0]
```

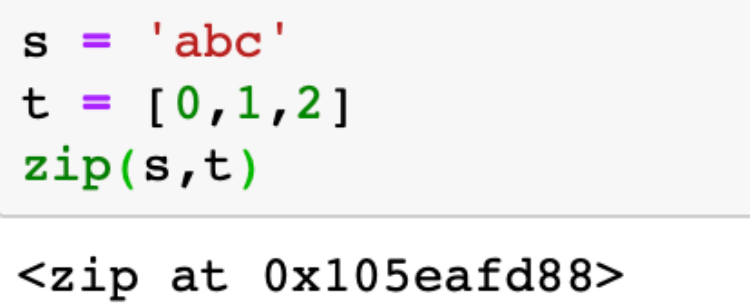
```
('x', 24)
```

# Zip

- zip is a built-in function that takes two or more sequences, and
- “zips” them into a list of tuples, where
- each tuple contains one element from each sequence

# Lists and Tuples

- Example,

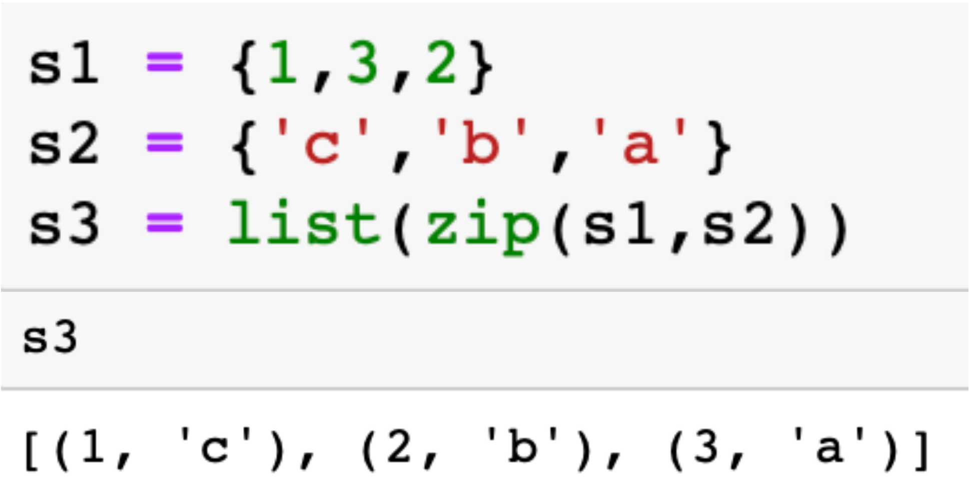


```
s = 'abc'
t = [0,1,2]
zip(s,t)
```

```
<zip at 0x105eafd88>
```

- The result is a list of tuples, where each tuple contains a character from the string and the corresponding element from the list

# Hashing from two arrays

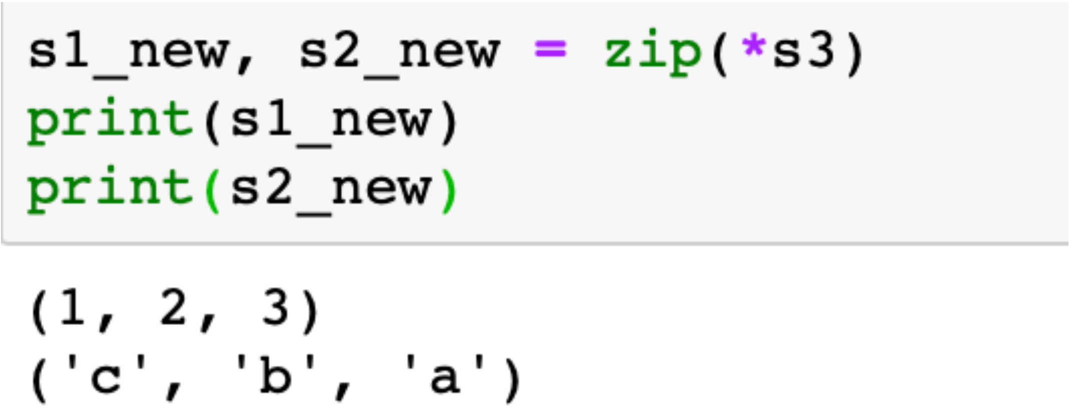


```
s1 = {1,3,2}
s2 = {'c','b','a'}
s3 = list(zip(s1,s2))
```

```
s3
```

```
[(1, 'c'), (2, 'b'), (3, 'a')]
```

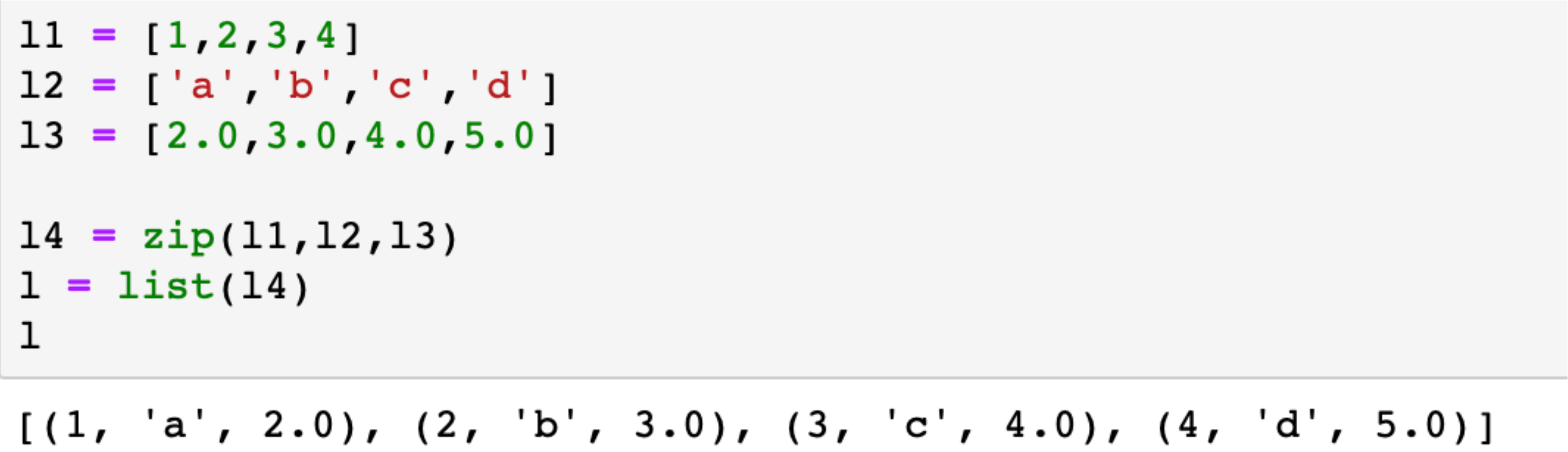
Unzip a list of tuples



```
s1_new, s2_new = zip(*s3)
print(s1_new)
print(s2_new)
```

```
(1, 2, 3)
('c', 'b', 'a')
```

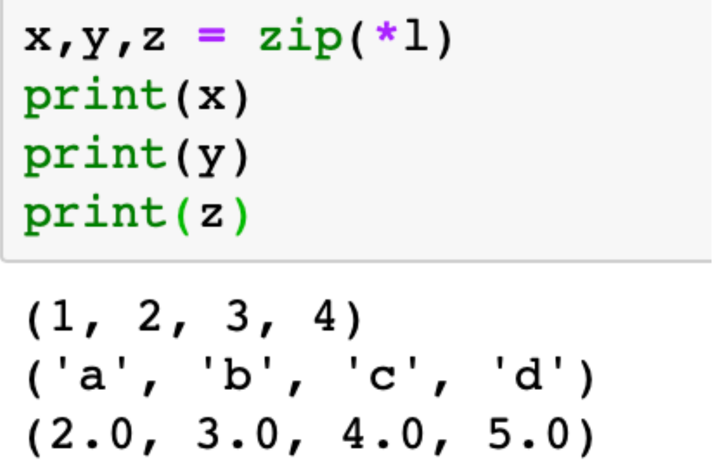
# Hashing from more than two arrays



```
l1 = [1,2,3,4]
l2 = ['a','b','c','d']
l3 = [2.0,3.0,4.0,5.0]

l4 = zip(l1,l2,l3)
l = list(l4)
l
```

```
[(1, 'a', 2.0), (2, 'b', 3.0), (3, 'c', 4.0), (4, 'd', 5.0)]
```

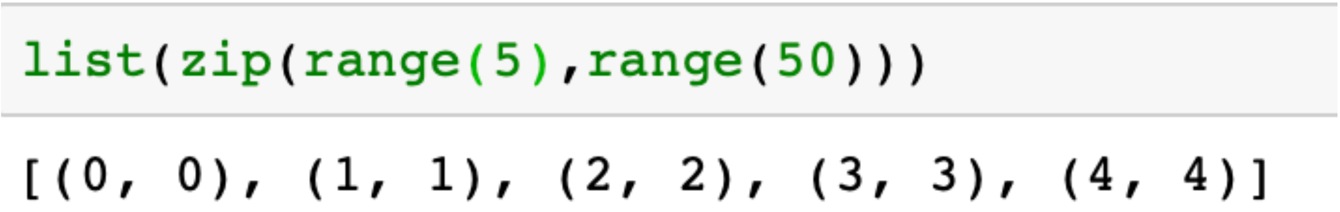
Unzip a list of tuples



```
x,y,z = zip(*l)
print(x)
print(y)
print(z)
```

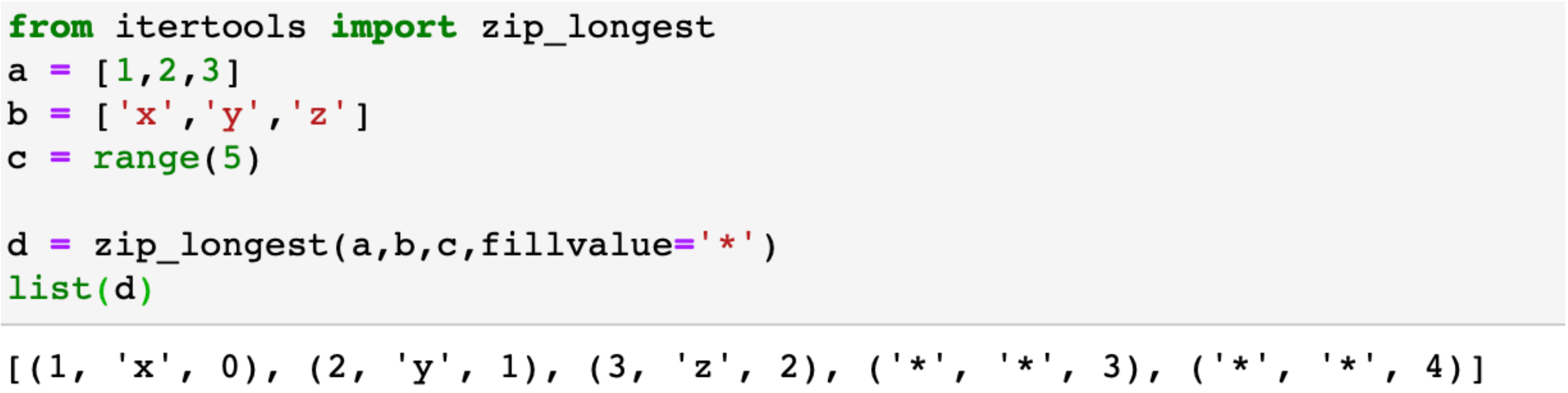
```
(1, 2, 3, 4)
('a', 'b', 'c', 'd')
(2.0, 3.0, 4.0, 5.0)
```

# Hashing from different sized arrays



```
list(zip(range(5),range(50)))
```

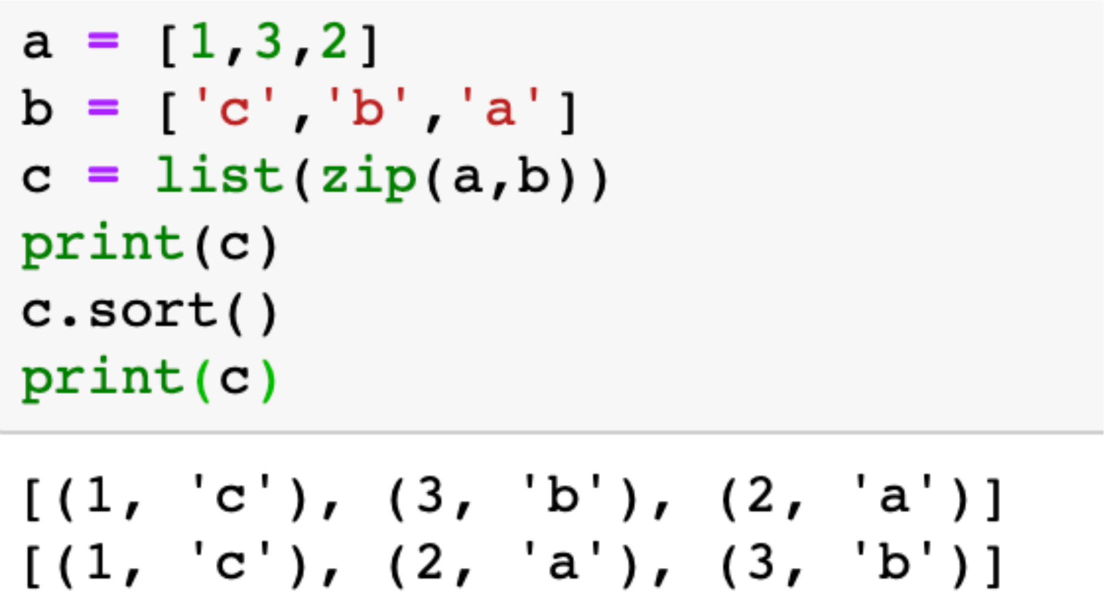
```
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
```



```
from itertools import zip_longest
a = [1,2,3]
b = ['x','y','z']
c = range(5)

d = zip_longest(a,b,c,fillvalue='*')
list(d)
```

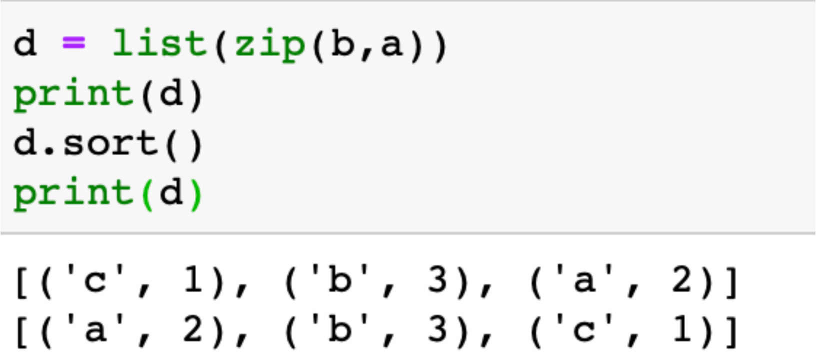
```
[(1, 'x', 0), (2, 'y', 1), (3, 'z', 2), ('*', '*', 3), ('*', '*', 4)]
```

# Sorting in Parallel



```
a = [1,3,2]
b = ['c','b','a']
c = list(zip(a,b))
print(c)
c.sort()
print(c)
```

```
[(1, 'c'), (3, 'b'), (2, 'a')]
[(1, 'c'), (2, 'a'), (3, 'b')]
```



```
d = list(zip(b,a))
print(d)
d.sort()
print(d)
```

```
[('c', 1), ('b', 3), ('a', 2)]
[('a', 2), ('b', 3), ('c', 1)]
```

# MSML 605
Files

# Introduction

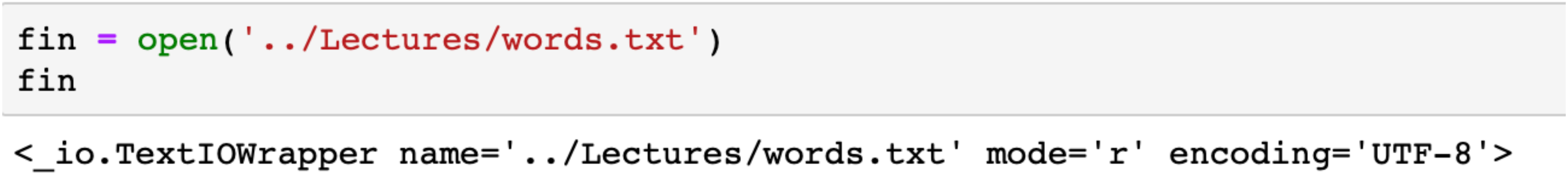
- Most of the programs written so far run for a short duration.
- Once the program ends, the data is gone
- If we want to see the results again we have to run the program again.

# Persistence

- Some programs run for a long time.
- They store data permanently
- The data is available even after the program ends.
- for example, operating systems and web servers
- One way to read and write data is using files.
- Another way to store data is using a database.

# Reading a File

- Using a built-in function ‘open’
- It takes the name of a file and returns a file object

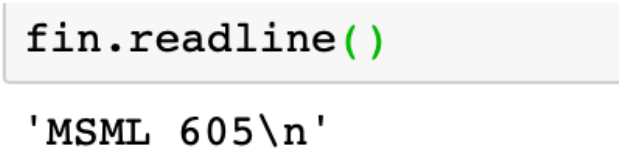


```
fin = open('../Lectures/words.txt')
fin
```

```
<_io.TextIOWrapper name='../Lectures/words.txt' mode='r' encoding='UTF-8'>
```

# Readline

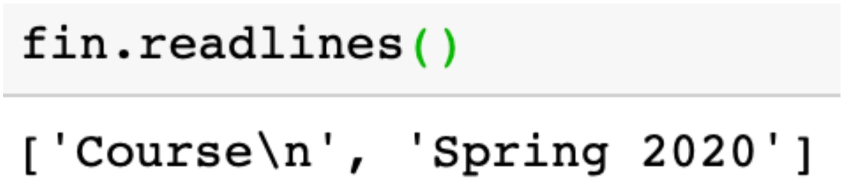
- It can read one line
  fin = open(‘words.txt’)
  fin.readline()



```
fin.readline()
```

```
'MSML 605\n'
```

- readlines() reads lines into a list



```
fin.readlines()
```

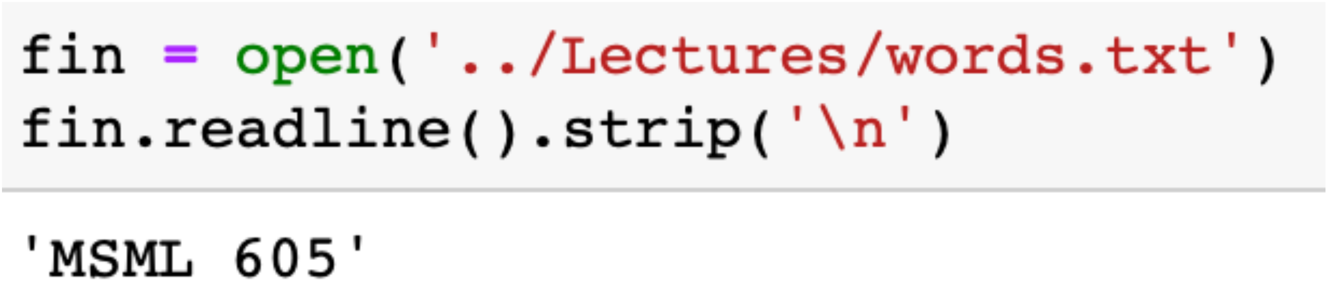
```
['Course\n', 'Spring 2020']
```

# End lines

- fin = open(‘words.txt’)
  fin.readline()

- Remove end line character

  fin.strip(“\n”)

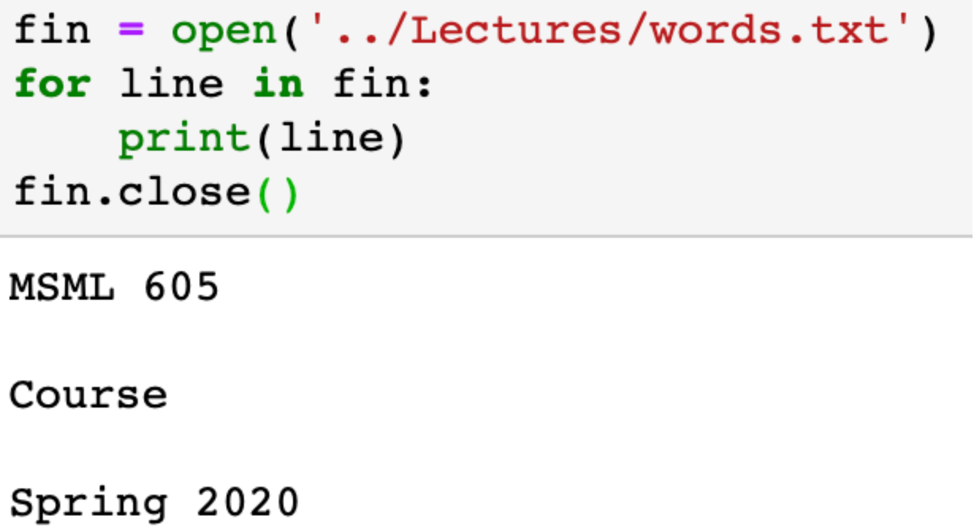


```
fin = open('../Lectures/words.txt')
fin.readline().strip('\n')
```

```
'MSML 605'
```

# File Traversal

-
  ```
  fin = open(‘words.txt’)
  for line in fin:
      print(line)
  ```



```
fin = open('../Lectures/words.txt')
for line in fin:
    print(line)
fin.close()
```

```
MSML 605

Course

Spring 2020
```

# Writing

- To write to a file, you have to open it with mode ‘w’ as a second parameter

  ```
  fout = open(‘output.txt’, ‘w’)
  ```

- If the file already exists, opening it in write mode clears out the old data and starts fresh

# Write to a File

```
line1 = “This is a ML class\n”
fout.write(line1)
line2 = “We Program in Python language\n”
fout.write(line2)
fout.close()
```

# Format Operator

- The argument of write has to be a string

- If we want to put other values in a file, we have to convert them to strings.

```
f = open(‘output.txt’, ‘w’)
x = 53
f.write(str(x))
```

- An alternative is to use the format operator, %

# Format Operator

- The argument of write is a string.
- If you want to write a string, you convert it to string first using

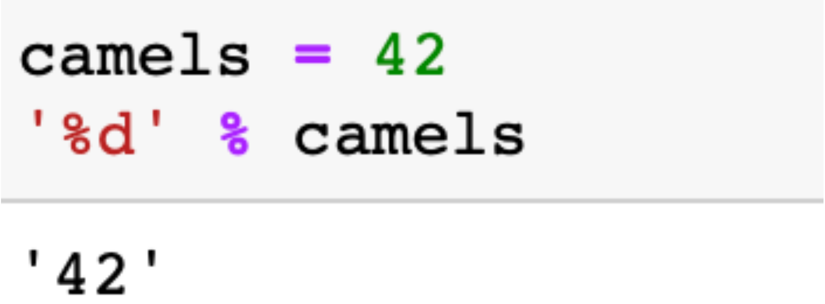
  str(<int value>)

  for example, str(4)

  converts int 4, to string.

# Format Sequence

- for example,
  the format sequence ‘%d’ means that the second operand should be formatted as an integer



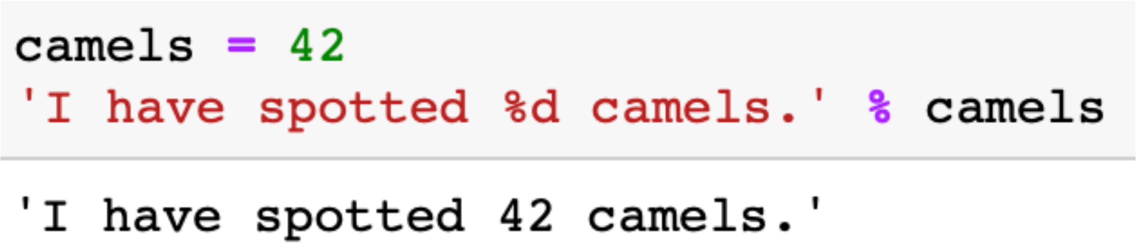
```
camels = 42
'%d' % camels
```

```
'42'
```

- The result is the string ‘42’

# More formatting

- A format sequence can appear anywhere in the string
- So you can embed a value in a sentence:

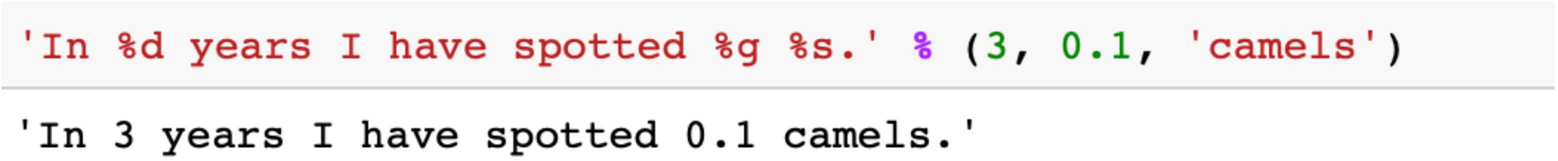


```
camels = 42
'I have spotted %d camels.' % camels
```

```
'I have spotted 42 camels.'
```

# More formatting

- For more than one format sequence in a string, the second argument is a tuple.
- Each format sequence with an element of the tuple, in order.
- Format Sequences used to format
  ‘%d’  an integer
  ‘%g’ a floating-point number
  ‘%s’ a string



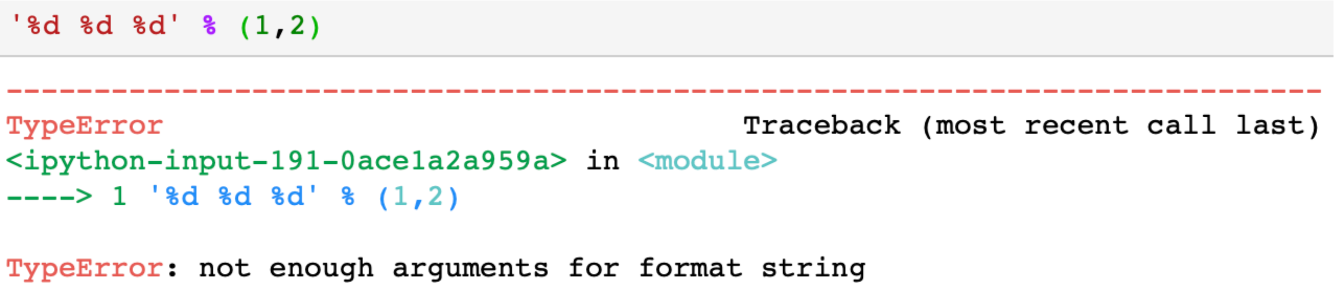
```
'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
```

```
'In 3 years I have spotted 0.1 camels.'
```

# Sequence formatting

- The number of elements in the tuple has to match the number of format sequences in the string
- Also, the types of the elements have to match the format sequences

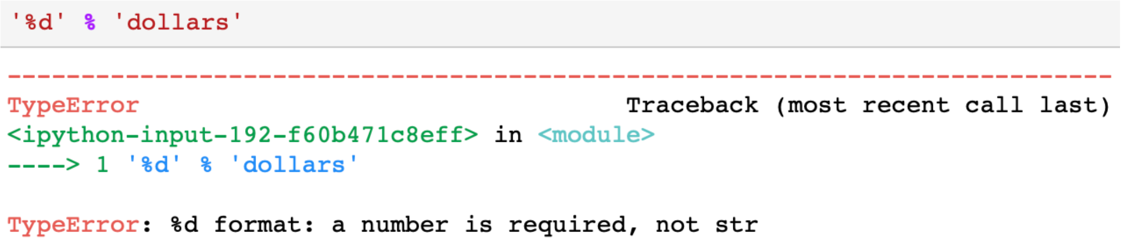
'%d %d %d' % (1,2)



```
'%d %d %d' % (1,2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-191-0ace1a2a959a> in <module>
----> 1 '%d %d %d' % (1,2)

TypeError: not enough arguments for format string
```

‘%d’ % ‘dollars'



```
'%d' % 'dollars'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-192-f60b471c8eff> in <module>
----> 1 '%d' % 'dollars'

TypeError: %d format: a number is required, not str
```

# Filenames and Paths

- import os
  os module provides functions for working with files and directories

```
>>> import os
>>> cwd = os.getcwd()
>>> print(cwd)
/Users/nayeem
```

- To find the absolute path to a file, you can use os.path.abspath

```
>>> os.path.abspath('words')
'/Users/nayeem/words'
```

# Filenames and Paths

- os.path.exists checks whether a file or directory exists:

```
>>> os.path.exists('words.txt')
False
```

- os.path.isdir checks whether it’s a directory:

```
>>> os.path.isdir('Documents')
True
```

- os.path.isfile checks whether it’s a file:

```
>>> os.path.isfile('test')
True
```

# Filenames and Paths

- os.listdir returns a list of the files ( and other directories) in the given directory:

```
>>> os.listdir(‘/users‘)
 ['.localized', 'Guest', 'nayeem', 'Shared']
```

- walk through a directory

```
import os

def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname,name)
        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

```
train_img_names = [os.path.join(training_path,f) for f in os.listdir(training_path) if f.endswith('.jpg')]
```

# Catching Exceptions

- If you try to open a file that doesn’t exist it will throw an error:

  ```
  fin = open('our_file')
  FileNotFoundError: [Errno 2] No such file or directory: 'our_file'
  ```

- If you don’t have permission to access a file:

  ```
  fout = open(‘/etc/passwd','w')
  PermissionError: [Errno 13] Permission denied: '/etc/passwd'
  ```

- If you try to open a directory for reading, you get:

  ```
  fin = open(‘/home')
  IsADirectoryError: [Errno 21] Is a directory: '/home'
  ```

# try and except

- There is an option using ‘try’ and ‘except’ so that the program does not halt when there is an error

```
try:
    fin = open('bad_file')
    for line in fin:
        print(line)
    fin.close()
except:
    print('Something went wrong')
```

- Python starts by executing the try clause.
- If all goes well, it skips the except clause and proceeds
- If an exception occurs, it jumps out of the try clause

# try and except

- There is an option using ‘try’ and ‘except’ so that the program does not halt when there is an error

```
try:
    fin = open('bad_file')
    for line in fin:
        print(line)
    fin.close()
except:
    print('Something went wrong')
```

- Python starts by executing the try clause.
- If all goes well, it skips the except clause and proceeds
- If an exception occurs, it jumps out of the try clause

# Pickling

- A pickle module is used to store Python objects in a database

```
import pickle
t = [1,2,3]
s = pickle.dump(t)
print(s)
t2 = pickle.load(s)
print(t2)
```

- Although the new object has the same value as the old, it is not the same object:

```
print(t==t2)     #True
print(t is t2)    # False
```