# Python Datastructures

Tuple, list, set, dict and others

# Tuples

- A tuple is a fixed-length, immutable (unchangeable) sequence of Python objects
- built-in functions that can be applied on tuples - isinstance( ), sorted( ), len( ), enumerate( ), reversed( )
- methods called on tuple object - index( ), count( )
- unpacking in loops
- use of "in"
- slicing
- ==, *, + supported
- modifying mutable elements of tuple allowed
- sorted( ) - operates on an iterable (list, str, tuple, set, dict keys, file) and returns a new list
- reversed( ) - returns a reverse iterator over the values of the given sequence (list, str, tuple)

# Tuple

```
t1 = 1,2,4; t1
isinstance(t1,tuple)
t2 = 1,'a',3.2,(3,4), complex(3,2), False,20,
bytes('abcdef','UTF-8');t2
[type(i) for i in t2]
```

- list to tuple

```
list1 = [4,5,3]
t3 = tuple(list1); t3
t3_3 = tuple(range(10))
#t3_3[0] = 11
t3_3
t4 = tuple('this is a new string line'); t4
print(t4)
t2[3]
t4[3:10]
```

- list is mutable so the list inside the immutable tuple can be modified

```
t5 = 1,2,[3,4,5],6
t5[2].append(7)
t5
```

- concatenation of tuples

```
t6 = t3+t5; t6
t3*4
t7 = t3*3; t7
list(enumerate(t7))
t1_1 = (1,2,4)

t1_2 = (2,1,4)
```

```
sorted(t1)==sorted(t1_2)
#use of tuple unpacking in for loops
t8 = ((1,2),(2,3),(3,4),(3,4))
for i,j in t8:
    #print("i=",i," j=",j)
    print(max(i,j))

### count occurrences of a particular value
t8.count((3,4))
len(t4)

t4[5:15]
t4.index('i')
list(reversed(t4))
't' in t4
's' not in t4
```

# List

- lists are variable-length and their contents can be modified in-place
- built-in functions that can be applied on lists - isinstance( ), sorted( ), len( ), enumerate( ), reversed( )
- methods called on list objects - clear( ), reverse( ), append( ), extend( ), insert( ), pop( ), remove( ), sort( )
- iteration using "in"
- tuple to list
- generator to list
- slicing
- ==, *, + supported
- in-place sorting: no copy made, more efficient than sorted function - sort( )
- sort method available only for list, sorted available for any iterable
- sort( ) vs sorted( ) and reverse( ) vs reversed( ) - sort and reverse are in-place and apply to list; sorted returns a new object and reversed returns a reverse iterator
- explore bisect, insort

# List

```
l3 = [1,'a',3.2,(3,4), complex(3,2), False,20]; l3
l3.clear()
l3.reverse()
```

- tuple to list

```
l4 = list(t8); l4
my_generator = range(10)
my_generator
```

- generator to list

```
l5 = list(my_generator); l5
l5.append('abcd'); l5
l5.insert(5,'def'); l5
```

- pop elements from an index position

```
l5.pop(5)
l5.pop(10)
l5
l5.insert(4,1)
l5.insert(4,1)
l5
```

- removes first occurrence of a particular element

```
l5.remove(1)
l5
6 in l5
```

- search in list is slower, if a lot of search needs to be done use dicts / sets

```
l6 = l5+l5
print(l6)
isinstance(l6,list)
l5*2
```

- use extend to add multiple elements

```
l6.extend(l4)
print(l6)
l5==l5
l5_1 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
l5_2 = [5, 2, 1, 3, 4, 0, 6, 7, 8, 9]
l5==l5_1
sorted(l5_1)==sorted(l5_2)

l7 = [ ]
for i in range(10):
  l7.append(i)
```

```
l7 = [ ]
for i in range(10):
    l7.append(i)
```

- list of lists

- flattening of lists - extend works by appending element one by one from an iterable (like a list)

```
l7 = [[1,2],[3,4]]
l8 = [ ]
for i in l7:
    l8.extend(i)
l8
```

- list of lists

- flattening of lists - DOES NOT work with append

```
l7 = [[1,2],[3,4]]
l8 = [ ]
for i in l7:
    l8.append(i)
l8
```

- extend function expects an iterable

```
l8.extend(1)
```

- in-place sorting: no copy made, more efficient than sorted function

- sort method available only for list, sorted available for any iterable

```
l9.sort(); l9

l10 =
['This','everything','something','thing','sing']; l10
l10.sort(); l10
l10.sort(key=str.lower); l10
l10.sort(key=len); l10
l10.sort(key=len,reverse=True); l10
#sort by 3rd character
#l10 =
['thing','everything','something','This','sing']
l10.sort(key=lambda x: x[2]); l10
l12=['everything', 'This', 'thing', 'something',
'sing','thim']
l12.sort(key=lambda x: x[2]); l12
```

# Use of zip( )

- Returns a zip object
- zip object can keep returning tuples where the i-th element comes from the i-th iterable argument
- Will continue returning until the shortest iterable in the argument sequence is exhausted and then it raises StopIteration
- Useful for constructing dict objects

```
l11 = [2,5,10,2,7,5,1,10]; l11
z1 = zip(l11,l12,l11)
list(z1)
type(z1)

lx = [3,7,5,9]; ly = ['a','b']
zz = zip(lx,ly); list(zz)


l13 = [('a',1),('d',2),('g',8)]

#special use of * to unpack zipped components
f1,f2 = zip(*l13)


f1,f2,f3 = zip(*l13)  # will give ValueError
```

# Dict

- keys for dict need to be immutable values - scalar types, tuples
- dictionary, key-value pair, hash map, associative array
- uses { }
- use key value to access elements
- use 'in' keys in for loops
- use del and pop to delete
- keys(), values(), update() available
- use zip to create dict from tuples/lists
- use dict() with zip() to create
- set and get default values available
- only == is supported;  -,+,* not supported

# Dict

```
d1 = {}
d1 = dict(); d1
d2 ={1:'orange',2:'banana',3:'apple'}; d2
```

- duplicate key results in the last element being used

```
d4 = {1:'orange',2:'banana',3:'apple',3:'pineapple'}; d4
d4_1={1: 'orange', 2: 'banana', 3: 'pineapple'}
d4==d4_1
d5 = {1:'orange',2:'banana',3:'apple',3:'pineapple','a':'apricot',3:'grape'}; d5
d5[2.4]='jackfruit'; d5

2.4 in d5

del d5[2.4]
d5.pop(2)
d5.keys()
list(d5.keys())
d5.values()
d6={'b':'strawberry',0:'apple'}
d5.update(d6)

t1 = range(5)
t2 = ('apple','jackfruit', 'grape', 'apricot', 'jackfruit')
list(zip(t1,t2))

d8 = dict(zip(t1,t2))
d8 = dict(zip(range(len(t2)),t2))
```

```
#use of default values in dictionaries
d8.get(4,'Anything here')
d8.get(6,'Anything here')

#set default has not effect if the key is present
for i in range(10):
    d8.setdefault(i,"NONE")
d8
```

# Set

- A set is an unordered collection of unique elements. similar to dicts but no values only keys
- Can be formed from an iterable
- set operations - union, intersection, difference, symmetric difference
- symmetric difference: all elements in either a or b not in both
- supports a lot of set operations
- supports update operations using update operator or add
- supports delete operations using clear, remove and pop
- does NOT support indexing
- only -,== supported; does not support +,*
- isintance( ), union( ), intersection( ), add( ), pop( ), remove( ), issubset( ), issuperset( ), isdisjoint( )

# Set

```
s1 = set(range(10))
isinstance(s1,set)
s2 = set([2,3,6,1])

s3 = set((2,3,4,7,2))

s4 = {1,4,2,6,1}

#set operations - union, intersection, difference, symmetric difference
a={1,2,3,4,5}
b={3,4,5,6,7,8}
a.union(b)
a | b

a.intersection(b)
a & b

a.add(1)
```

```
a-b
a -=b; a

#symmetric difference: all elements in either a or b not in both
a ^ b

a.issubset(b)
a.issuperset(b)
a.isdisjoint(b)

c = {2,3,4,1,7,5}
a==c

#update on union operator
a |= b
a

a.pop()
a.remove(6)
```

# Usage of the Basic Python Data-structures

- tuples used to
  - return multiple values from functions;
  - unpack multiple values in for loop and similar places
  - hold constant values in functions
- dictionaries used to
  - read various key-value pair data such as markup datasets (XML, JSON, etc.)
  - supply values for replacement in pandas dataframes
- list used in a very general sense in most places
  - list comprehension is most popular and widely used in
    - data transformation (to hold values which is passed as parameters to pandas objects)
    - numerical computations (linear algebra)
- set usage is similar to dicts: used to hold unique values during data transformation