

# Introduction to Python

## **Tuples and Dictionaries**

# Topics

- 1) Tuples
- 2) Tuple unpacking
- 3) List of tuples, List of lists(2D lists)
- 4) enumerate()
- 5) Dictionaries
- 6) Iterating over a dictionary

# Tuples

Tuples are in many ways similar to lists, but they are defined with parentheses rather than square brackets:

```
t = (1, 2, 3)
```

They can also be defined without any brackets at all:

```
t = 1, 2, 3  
print(t, type(t))  # (1, 2, 3) tuple
```

# Tuples

Like the lists, tuples have a length and individual elements can be extracted using square-bracket indexing. Slicing is also supported.

```
t = (1, 2, 3)
print(len(t))      # 3
print(t[0])        # 1
print(t[0:2])      # [1, 2]
```

# Tuples

Unlike lists, tuples are *immutable*: this means that once they are created, their size and contents cannot be changed:

```
lst = [1, 2, 3]
lst[1] = 5           # ok! a list is mutable
t = (1, 2, 3)
t[0] = 1             # error! a tuple is immutable
t.append(10)         # error! a tuple is immutable
```

*Note: a tuple takes less memory than a list and can be generally manipulated faster.*

# Tuple Unpacking

Tuple unpacking is an assignment feature that assigns right hand side of values into left hand side. In packing, we put values into a new tuple while in unpacking we extract those values into a single variable.

This is packing values into a variable.

```
student = ("Mike Smith", 3.2)
```

This is unpacking values from a variable.

```
name, gpa = student  
print(name, gpa)      # Mike Smith 3.2
```

# Tuple Unpacking

Tuple unpacking can be used to do parallel assignment.

```
a = 1
b = 2
a, b = b, a    # parallel assignment
print(a, b)    # 2 1
```

Note: In Java, you would need a temporary variable for this.

# list(), tuple()

Converting between sequences can be done using the appropriate constructors: list(), tuple().

```
lst = [1, 2, 3]
```

```
t = tuple(lst)          # (1, 2, 3)
```

```
s = (1, 2, 3)
```

```
lst2 = list(s)          # [1, 2, 3]
```



# Iterating over tuples

Like lists, we can iterate over a tuple.

```
lst = [1, 2, 3]
```

```
for x in lst:
```

```
    print(x, end=" ")
```

```
1 2 3
```

```
t = (1, 2, 3)
```

```
for y in t:
```

```
    print(y, end=" ")
```

```
1 2 3
```

# List of Tuples

We can form a list of tuples(or of lists) and iterate over them.

There are three ways to iterate over a list of tuples.We will show code for all three methods in the next few slides.

## Method 1) Using nested loop:

```
lst = [("Mike", 3.2), ("Sarah", 3.6), ("Jack", 2.8)]
```

```
for student in lst:
```

```
    for data in student:
```

```
        print(data)
```

Mike

3.2

Sarah

3.6

Jack

2.8

Note:We can use a list of lists in this example.

But this is a case where a list of tuples is a bit easier to use.

# List of Tuples

We can form a list of tuples(or of lists) and iterate over them.

Method 2) Using []:

```
lst = [("Mike", 3.2), ("Sarah", 3.6), ("Jack", 2.8)]  
for student in lst:  
    print(student[0], student[1])
```

Mike 3.2

Sarah 3.6

Jack 2.8

# List of Tuples

We can form a list of tuples(or of lists) and iterate over them.

Method 3) Using tuple unpacking. This is most pythonic method.

```
lst = [("Mike", 3.2), ("Sarah", 3.6), ("Jack", 2.8)]  
for name, gpa in lst: # tuple unpacking  
    print(name, gpa)
```

Mike 3.2

Sarah 3.6

Jack 2.8

# Average GPA

Using tuple unpacking to compute the average.

```
lst = [("Mike", 3.2), ("Sarah", 3.6), ("Jack", 2.8)]  
s = 0  
for name, gpa in lst: # tuple unpacking  
    s += gpa  
ave = s / len(lst)
```

# enumerate()

It is useful to have access to the index of elements when iterating over a list or 2D list. For example, if we need to modify elements of a list, having access to the indices is useful.

The `enumerate()` function adds an index to elements of the list and returns it.

```
lst = ['bread', 'milk', 'ham']  
for item in enumerate(lst):  
    print(item)
```

```
(0, 'bread')
```


```
(1, 'milk')
```

```
(2, 'ham')
```

# enumerate()

Unpacking from enumerate().

```
lst = ['bread', 'milk', 'ham']  
for index, value in enumerate(lst):  
    print(index, value)
```



0 bread

1 milk

2 ham

This is a tuple!

# enumerate()

enumerate() is helpful when we want to modify our list.

```
lst = ['bread', 'milk', 'ham']  
for index, value in enumerate(lst):  
    if value == 'bread':  
        lst[index] = 'butter'  
print(lst)
```

Output:

```
['butter', 'milk', 'ham']
```



# Dictionaries

Python lists are useful but in some applications, it is nice to have a different indexing scheme than the integers. For example, consider a database of students' names and their grades:

Mike Smith: [70, 81, 84]

Sarah Johnson: [88, 71, 85]

...

Suppose that this database has hundreds of records. It is hard to access these students' grades using 0-based integer indexing.

Python dictionaries allow "values" to be accessed by meaningful "keys". In the example above, we can access the database of grades by name(keys) instead of integer index.

# Dictionaries

Dictionaries are extremely flexible mappings of keys to values, and form the basis of much of Python's internal implementation.

They can be created via a comma-separated list of key:value pairs within curly braces. The "keys" must be distinct.

```
data = {"Mike":3.1, "Sarah":3.6, "John":3.4}
print(data["Mike"])           # 3.1
print(data["John"])           # 3.4
print(len(data))              # 3
```



# Dictionaries

Modifying dictionary.

```
data = {"Mike":3.1, "Sarah":3.6, "John":3.4}
data['Mike'] = 3.2
print(data)      # {'Mike': 3.2, 'Sarah': 3.6, 'John': 3.4}
data['Sarah'] += 0.2
print(data)      # {'Mike': 3.2, 'Sarah': 3.8, 'John': 3.4}
```

# Dictionaries

The keys and values of dictionaries can be different types. However, a dictionary key must be immutable(int, float, bool, str, tuple). A dictionary value can be any object.

```
misc = {2:5, (3, 5):4.5, True:[1], "a":(1, 5)}  
print(misc[2])          # 5  
print(misc[True])       # [1]  
print(misc[(3, 5)])     # 4.5  
print(misc["a"])        # (1, 5)
```

# Membership Operations

By default, membership operations **checks keys** of a dictionary.

```
scores = {'Mike':5, 'John':2, 'Sarah':4}
```

```
print('Mike' in scores)           # True
```

```
print(5 in scores)                # False
```

```
print('Michele' not in scores)    # True
```

This will allow use to loop through a dictionary.

# Iterating over keys a dictionary

It is easy to iterate over keys of the dictionary. The default loop iterates over the keys.

```
grades = {"Mike":3.1, "Sarah":3.6, "John":3.4}  
for x in data:  
    print(x, end=" ")
```

Output:

Mike Sarah John

# Iterating over keys a dictionary

The following compute the average GPA.

```
grades = {"Mike":3.1, "Sarah":3.6, "John":3.4}
sum = 0
for student in grades:
    sum += grades[student]
average = sum/len(grades)
```



# Example of a Use for Dictionaries

One use of a dictionary is keep track of frequency count.

```
words = ['baby', 'shark', 'do', 'do', 'do', 'do', 'do', 'do']
```

```
frequency = {}
```

```
for word in words:
```

```
    if word not in frequency:
```

```
        frequency[word] = 0
```

```
    frequency[word] += 1
```

```
print(frequency)
```

We will use this code to do word frequency analysis of the works of Shakespeare!

Output:

```
{'baby': 1, 'shark': 1, 'do': 6}
```

# References

- I) Vanderplas, Jake, A Whirlwind Tour of Python, O'reilly Media.