Department of Computer Science and Engineering
PES University, Bangalore, India

# Python For Computational Problem Solving (UE22CS151A)

**Mr. Prakash C O**
**Asst. Professor,**
**Dept. of CSE, PESU**
**coprakasha@pes.edu**

---

## Dictionary - dict():

Dictionary is more generally known as an associative array. A dictionary consists of a collection of key-value pairs. Each key-value pair maps the key to its associated value.

- A dictionary is a collection which is ordered, mutable, indexed through keys and does not allow duplicates.
- Dictionaries are written with curly brackets, and they have key-value pairs.
- Each key-value pair in a Dictionary is separated by a colon(:), whereas many key:value pairs are separated by a comma.

Keys of a Dictionary must be unique and immutable, but the values associated with keys can be repeated and be of any type.

**Dictionaries and lists share the following characteristics:**

- Both are mutable.
- Both are dynamic. They can grow and shrink as needed.
- Both can be nested. A list can contain another list. A dictionary can contain another dictionary. A dictionary can also contain a list, and vice versa.

**Dictionaries differ from lists primarily in how elements are accessed:**

- List elements are accessed by their position in the list, via indexing.
- Dictionary elements are accessed via keys.

**Note:**

- As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.
- Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”.

**Creating Dictionary**

Approach 1:

You can define a dictionary by enclosing a comma-separated list of key-value pairs in curly braces ({}). A colon (:) separates each key from its associated value:

In [1]:
```
cardict = {
```

```
            "brand" : "Ford",
            "model" : "Mustang",
            "year"  : 1964,
            "colors":["red", "blue", "yellow", "grey"]
          }
print(cardict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'colors': ['red', 'blue', 'yellow',
'grey']}
```

In [1]:
```
weekdays_dict = {
                  0:'Monday',
                  1:'Tuesday',
                  2:'Wednesday',
                  3:'Thursday',
                  4:'Friday',
                  5:'Saturday',
                  6:'Sunday'
                }
print(weekdays_dict)
```

```
{0: 'Monday', 1: 'Tuesday', 2: 'Wednesday', 3: 'Thursday', 4: 'Friday', 5: 'Saturday',
6: 'Sunday'}
```

Approach 2:

You can also construct a dictionary with the dict() constructor. The argument to dict() should be a sequence of key-value pairs. A list of tuples works well for this:

In [16]:
```
# list of tuples as an argument
cardict = dict([
                ("brand","Ford"),
                ("model","Mustang"),
                ("year",1964),
                ("colors",["red", "blue", "yellow", "grey"])
])
print(cardict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'colors': ['red', 'blue', 'yellow',
'grey']}
```

In [17]:
```
# tuple of tuples as an argument
cardict = dict( (
                 ("brand","Ford"),
                 ("model","Mustang"),
                 ("year",1964),
                 ("colors",["red", "blue", "yellow", "grey"])
                ) )
print(cardict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'colors': ['red', 'blue', 'yellow',
'grey']}
```

In [3]:
```
# set of tuples as an argument
cardict = dict( {
                 ("brand","Ford"),
                 ("model","Mustang"),
                 ("year",1964),
                 ("colors","red")
                } )
print(cardict)
```

```
{'colors': 'red', 'model': 'Mustang', 'brand': 'Ford', 'year': 1964}
```

In [2]:
```
# list of tuples as an argument
weekdays_dict = dict([(0,'Monday'),(1,'Tuesday'),(2,'Wednesday'),
```

```
                      (3,'Thursday'),(4,'Friday'),(5,'Saturday'),
                      (6,'Sunday')])
print(weekdays_dict)
```

```
{0: 'Monday', 1: 'Tuesday', 2: 'Wednesday', 3: 'Thursday', 4: 'Friday', 5: 'Saturday',
6: 'Sunday'}
```

Approach 3:

If the dictionary keys are simple strings, they can be specified as keyword arguments.

In [4]:
```
MLB_team = dict(
               Colorado = 'Rockies',
               Boston = 'Red Sox',
               Minnesota = 'Twins',
               Milwaukee = 'Brewers',
               Seattle = 'Mariners',
               Year = 2022
              )
print(MLB_team)
```

```
{'Colorado': 'Rockies', 'Boston': 'Red Sox', 'Minnesota': 'Twins', 'Milwaukee': 'Brewer
s', 'Seattle': 'Mariners', 'Year': 2022}
```

In [2]:
```
# Example 2
Knowledge = {'Frank': {'Python', 'Perl'},
             'Guido': {'Python'},
             'Monica': {'C', 'C++'}}
print(Knowledge)
```

```
{'Frank': {'Perl', 'Python'}, 'Guido': {'Python'}, 'Monica': {'C', 'C++'}}
```

#### Building a Dictionary Incrementally

Defining a dictionary using curly braces and a list of key-value pairs, as shown above, is fine if you know all the keys and values in advance. But what if you want to build a dictionary on the fly?

You can start by creating an empty dictionary, which is specified by empty curly braces. Then you can add new keys and values one at a time:

In [1]:
```
person_dict = {}  # empty dictionary

person_dict['fname'] = 'Raj'      #add items one by one
person_dict['lname'] = 'Patil'
person_dict['age'] = 51
person_dict['spouse'] = 'Raksha'
person_dict['children'] = ['Vinay', 'Vijetha']
person_dict['pets'] = {'dog': 'Sam', 'cat': 'Sweety'}

print(person_dict)
```

```
{'fname': 'Raj', 'lname': 'Patil', 'age': 51, 'spouse': 'Raksha', 'children': ['Vinay',
'Vijetha'], 'pets': {'dog': 'Sam', 'cat': 'Sweety'}}
```

This example exhibits another feature of dictionaries: the values contained in the dictionary don't need to be the same type. In person, some of the values are strings, one is an integer, one is a list, and one is another dictionary.

## Dictionary Items

- Dictionary items are presented in (key:value) pairs, and can be referred to by using the key name.

## A dictionary is a data structure with zero or more elements with the following attributes.

- Keys are unique(within one dictionary). Duplicate keys are not allowed.
- Keys are immutable – cannot be changed
- Keys are hashable - Key value pairs are stored at the hashed location in some way
- Dictionary is ordered and indexable based on the key – key could be a string, an integer or a tuple or any immutable type
- An empty dictionary is created by using {} or by the constructor dict()
- We can extract
    - keys using the method dict.keys() and
    - values using the method dict.values().

  Both these return iterable view objects.
- We can iterate through a dict – we actually iterate through the keys.

#### Creating empty dictionary

In [25]:
```
d1 = {}      # Creating empty dictionary by using empty pair of {}
d2 = dict() # Creating empty dictionary by calling dictionary constructor
print(d1,type(d1))
print(d2,type(d2))
```

```
{} <class 'dict'>
{} <class 'dict'>
```

#### Dictionary is ordered and indexed by keys

- Dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys.
- Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key.

In [ ]:
```
d3 = {  1       : "Hi",
        2       : "Hello",
       (3,4,[]): "How r u?" }  # TypeError: unhashable type: 'list'

print(d3)
#print(hash((3,4,[])))
```

In [ ]:
```
t = ([1,2,3],4)
my_dict = {'name':'John', t:'values'} # TypeError: unhashable type: 'list'
print(my_dict)
```

In [30]:
```
my_dict = {'name': 'John', tuple([1,2,3]):'values'}
print(my_dict)
```

```
{'name': 'John', (1, 2, 3): 'values'}
```

#### Technical Note: Why does the error message say “unhashable”?

- Technically, it is not quite correct to say an object must be immutable to be used as a dictionary key. More precisely, an object must be hashable, which means it can be passed to a hash function.
- A hash function takes data of arbitrary size and maps it to a relatively simpler fixed-size value called a hash value (or simply hash), which is used for table lookup and comparison.

Python's built-in hash() function returns the hash value for an object which is hashable, and raises an exception for an object which isn't.

**Keys are unique(within one dictionary). Duplicate keys are not allowed.**

- Dictionaries cannot have two items with the same key
- Duplicate key values will overwrite existing values

In [9]:
```
cardict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(cardict)
# Print the number of items in the dictionary:
print(len(cardict))
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
3
```

**The values in dictionary items can be of any data type**

In [ ]:
```
cardict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
```

**Dictionary is Iterable**

In [12]:
```
cardict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
for key in cardict:  #we actually iterate through the keys
    print(key,"->",cardict[key])
```

```
brand -> Ford
electric -> False
year -> 1964
colors -> ['red', 'white', 'blue']
```

**Dictionary is mutable**

In [6]:
```
cardict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
cardict["color"] = "red"
print(cardict)
cardict["model"] = "ecosports"
print(cardict)
```

```
{'brand': 'Ford', 'model': ('Mustang',), 'year': 1964, 'color': 'red'}
{'brand': 'Ford', 'model': ('ecosports',), 'year': 1964, 'color': 'red'}
```

## Dictionary Methods

Python has a set of built-in methods that you can use on dictionaries.

| Method | Description |
|---|---|
| clear() | Removes all the elements from the dictionary |
| copy() | Returns a copy of the dictionary |
| fromkeys() | Returns a dictionary with the specified keys and value |
| get() | Returns the value of the specified key |
| items() | Returns a list containing a tuple for each key value pair |
| keys() | Returns a list containing the dictionary's keys |
| pop() | Removes the element with the specified key |
| popitem() | Removes the last inserted key-value pair |
| setdefault() | Returns the value of the specified key. If the key does not exist: insert the key, with the specified value |
| update() | Updates the dictionary with the specified key-value pairs |
| values() | Returns a list of all the values in the dictionary |

**clear() - clears a dictionary.**

d.clear() empties dictionary d of all key-value pairs:

In [14]:
```
d = {'a': 10, 'b': 20, 'c': 30}
print(d)
d.clear()
print(d)
```

```
{'a': 10, 'b': 20, 'c': 30}
{}
```

**get(key [, default])**

Returns the value for a key if it exists in the dictionary.

The Python dictionary .get() method provides a convenient way of getting the value of a key from a dictionary without checking ahead of time whether the key exists, and without raising an error.

d.get(key) searches dictionary d for key and returns the associated value if it is found. If key is not found, it returns None:

In [15]:
```
d = {'a': 10, 'b': 20, 'c': 30}
print(d.get('b')) #If key is found, it returns the associated value
print(d.get('z')) #If key is not found, it returns None
```

```
20
None
```

If key is not found and the optional default argument is specified, that value is returned instead of None:

In [10]:
```
d = {'a': 10, 'b': 20, 'c': 30}
print(d.get('z', 3))
```

```
3
```

In [6]:
```
# get() - a conventional method to access a value for a key.
thisdict = {
```

```
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
print(thisdict.get("brand"))
```

```
Ford
```

#### values() - returns a list of values in a dictionary

values() method returns a view object containing a list of all the values in the dictionary

In [8]:
```
cardict = {
  "brand"   : "Ford",
  "electric": False,
  "year"    : 1964,
  "colors"  : ["red", "white", "blue"]
}
print(cardict.values())
#print(list(cardict.values()))

for value in cardict.values():
    print(value, end=' ')
```

```
dict_values(['Ford', False, 1964, ['red', 'white', 'blue']])
Ford False 1964 ['red', 'white', 'blue']
```

#### keys() - returns a list of keys in a dictionary.

keys() method returns a view object containing a list of all the keys in the dictionary

In [21]:
```
cardict = {
  "brand"   : "Ford",
  "electric": False,
  "year"    : 1964,
  "colors"  : ["red", "white", "blue"]
}
print(cardict.keys())
print(list(cardict.keys()))
#for key in cardict.keys():
#    print(key, end=' ')
```

```
dict_keys(['brand', 'electric', 'year', 'colors'])
['brand', 'electric', 'year', 'colors']
```

#### pop(key[, default])

Removes an item from a dictionary with the specified key.

If key is present in d, d.pop(key) removes key and returns its associated value.

In [22]:
```
# pop() method - Removes the element with the specified key
cardict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
print(cardict.pop("electric"))
print(cardict)
```

```
False
{'brand': 'Ford', 'year': 1964, 'colors': ['red', 'white', 'blue']}
```

pop(key) raises a KeyError exception if key is not in dictionary.

In [ ]:
```
print(cardict.pop("model"))  # KeyError: 'model'
print(cardict)
```

If key is not in dictionary, and the optional default argument is specified, then that value is returned, and no exception is raised.

In [25]:
```
print(cardict.pop("model",'ecosports'))
print(cardict)
```

```
ecosports
{'brand': 'Ford', 'year': 1964, 'colors': ['red', 'white', 'blue']}
```

**popitem()**

Removes the last inserted key-value pair from a dictionary and returns it as a tuple.

d.popitem() removes the last key-value pair added to d and returns it as a tuple:

In [26]:
```
# popitem() removes the last inserted key-value pair
cardict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
print(cardict.popitem())
print(cardict)
print(cardict.popitem())
print(cardict)
```

```
('colors', ['red', 'white', 'blue'])
{'brand': 'Ford', 'electric': False, 'year': 1964}
('year', 1964)
{'brand': 'Ford', 'electric': False}
```

If d is empty, d.popitem() raises a KeyError exception:

In [ ]:
```
d = {}
print(d.popitem()) #KeyError: 'popitem(): dictionary is empty'
```

**items() Method**

The items() method returns a view object. The view object contains the key-value pairs of the dictionary, as tuples in a list.

The first item in each tuple is the key, and the second item is the key's value.

The view object will reflect any changes done to the dictionary, see example below.

In [19]:
```
# When an item in the dictionary changes value,
  # the view object also gets updated:

cardict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

d_items = cardict.items()
cardict["year"] = 2018
print(d_items)
```

```
print(list(d_items))
print(cardict)
```

```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 2018)])
[('brand', 'Ford'), ('model', 'Mustang'), ('year', 2018)]
{'brand': 'Ford', 'model': 'Mustang', 'year': 2018}
```

**setdefault(key[, default])**

The setdefault() method returns the value of the item with the specified key.

If the key does not exist, insert the key, with the specified value.

Syntax: dictionary.setdefault(keyname, value)

Parameter Values:

| Parameter | Description |
|---|---|
| *keyname* | Required. The keyname of the item you want to return the value from |
| *value* | Optional.<br>If the key exist, this parameter has no effect.<br>If the key does not exist, this value becomes the key's value<br>Default value None |

In [6]:
```
# Get the value of the "model" item:

cardict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
# The key model exists in cardict, then the default value
#  Bronco has no effect.
value = cardict.setdefault("model", "Bronco")
print(value)
print(cardict)
```

```
Mustang
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

In [8]:
```
''' Get the value of the "color" item, if the "color" item does not exist,
    insert "color" with the value "white" '''
cardict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
# The key color does not exist in cardict, then the default
#  value white becomes the key's value.
value = cardict.setdefault("color", "white")
print(value)
print(cardict)
```

```
white
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'white'}
```

**fromkeys(keys[, value])**

- keys are required - An iterable specifying the keys of the new dictionary
- value is optional - The value same for all keys. Default value is None

The fromkeys() method returns a dictionary with the specified keys and the specified value.

In [20]:
```
t1 = (1,2,3)
my_dict = dict.fromkeys(t1)
print(my_dict)
```

```
{1: None, 2: None, 3: None}
```

In [21]:
```
t1 = (1,2,3)
my_dict = dict.fromkeys(t1,'Hello')
print(my_dict)
```

```
{1: 'Hello', 2: 'Hello', 3: 'Hello'}
```

**update(obj)**

Merges a dictionary with another dictionary or with an iterable of key-value pairs.

If obj is a dictionary, d.update(obj) merges the entries from obj into d. For each key in obj:

- If the key is not present in d, the key-value pair from obj is added to d.
- If the key is already present in d, the corresponding value in d for that key is updated to the value from .

Here is an example showing two dictionaries merged together:

In [28]:
```
d1 = {'a': 10, 'b': 20, 'c': 30}
d2 = {'b': 200, 'd': 400}

d1.update(d2)
print(d1)
```

```
{'a': 10, 'b': 200, 'c': 30, 'd': 400}
```

In this example, key 'b' already exists in d1, so its value is updated to 200, the value for that key from d2. However, there is no key 'd' in d1, so that key-value pair is added from d2.

### Turn Lists into Dictionaries

The zip() function take iterables (can be zero or more), makes iterator that aggregates elements based on the iterables passed, and returns an iterator of tuples.

In [6]:
```
dishes = ["pizza", "sauerkraut", "paella", "hamburger"]
countries = ["Italy", "Germany", "Spain", "USA"]
country_specialities_dict = dict(zip(countries, dishes))
print(country_specialities_dict)

lt=zip(countries, dishes)
print(lt)
print(list(lt))
```

```
{'Italy': 'pizza', 'Germany': 'sauerkraut', 'Spain': 'paella', 'USA': 'hamburger'}
<zip object at 0x000001335738E080>
[('Italy', 'pizza'), ('Germany', 'sauerkraut'), ('Spain', 'paella'), ('USA', 'hamburge
r')]
```

**Exercise Problems:**

1.Write a program to find the frequency of words in a given string and store word and its count as key:value pairs in dictionary.

In [1]:
```
#string = "how much wood would a wood chuck chuck if a wood chuck could chuck wood"
#output: {'how': 1, 'much': 1, 'wood': 4, 'would': 1, 'a': 2, 'chuck': 4, 'if': 1, 'coul

str1="how much wood would a wood chuck chuck if a wood chuck could chuck wood"
```

```
words_list = str1.split()
wordcount_dict = {}
for word in words_list:
    if word not in wordcount_dict:
        wordcount_dict[word] = 0
    wordcount_dict[word] += 1

print(wordcount_dict)
```

```
{'how': 1, 'much': 1, 'wood': 4, 'would': 1, 'a': 2, 'chuck': 4, 'if': 1, 'could': 1}
```

2.What is the output?

In [1]:
```
d={1:2, 4:8, 3:4, 2:3}
print(d[d[d[d[1]]]])
```

```
8
```

In [9]:
```
d={1:2, 4:8, 3:1, 2:4}
print(d[d[d[d[3]]]])
```

```
8
```

In [1]:
```
d={1:4,4:6,6:10,10:'hello'}
print(d[d[d[d[1]]]])
```

```
hello
```

3.What is the output?

In [1]:
```
a={}
for i in range(5):
    a[i%2==0]=i
    print(a)
#print(a)
```

```
{True: 0}
{True: 0, False: 1}
{True: 2, False: 1}
{True: 2, False: 3}
{True: 4, False: 3}
```

4.What is the output?

In [14]:
```
d={ True:1, 1:2 }
print(d)
print(hash(True))
print(hash(1))
```

```
{True: 2}
1
1
```

Only one element in the dictionary, because keys (True and 1) have the same hash value. The value 1 is

5.What is the output?

In [2]:
```
s = 'My name is Raja Kumar'
words_list=s.split()
dict1={}
for word in words_list:
    dict1[len(word)]=word
print(dict1)
print(dict1[4])
```

```
{2: 'is', 4: 'Raja', 5: 'Kumar'}
Raja
```

In [1]:
```
d = {"john":40, "peter":45}
print(d)
```

```
{'john': 40, 'peter': 45}
```

6.What will be the output of the following Python code snippet?

In [2]:
```
d1 = {"john":40, "peter":45}
d2 = {"john":466, "peter":45}
print(d1 > d2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [2], in <cell line: 3>()
      1 d1 = {"john":40, "peter":45}
      2 d2 = {"john":466, "peter":45}
----> 3 print(d1 > d2)

TypeError: '>' not supported between instances of 'dict' and 'dict'
```

Explanation: Arithmetic > operator cannot be used with dictionaries.

## Efficient Use of Data Structures and Algorithms

An essential aspect of code optimization is understanding data structures and algorithms. The right choices can lead to significant improvements in performance.

### Choosing the right data structures

Selecting the proper data structures can significantly impact the performance of your code. For example, when performing lookups, a set or a dict can be much faster than searching through a list.

In [1]:
```
# Using a list for membership testing
list_example = [1, 2, 3, 4, 5]
if 5 in list_example:
    print("Found in list!")

# Using a set for much faster membership testing
set_example = {1, 2, 3, 4, 5}
if 5 in set_example:
    print("Found in set!")
```

```
Found in list!
Found in set!
```

Always choose the most appropriate data structure for your use-case, taking into account factors such as time complexity, space complexity, and built-in operations.

## Let us solve some problems and choose the right data structures.

- The input to all these problems is a string having # of lines.
- Each line has a language name, a writer's name and his work.
- We can split the string based on new line and split each of these lines based on white space and capture whatever is required.

In [11]:
```
all = """sanskrit kalidasa shakuntala
```

```
english r_k_narayan malgudi_days
kannada kuvempu ramayanadarshanam
sanskrit bhasa swapnavasavadatta
kannada kuvempu malegalalli_madumagalu
english r_k_narayan dateless_diary
kannada karanta chomanagudi
sanskrit baana harshacharita
kannada karanta sarasatammana_Samadhi
sanskrit kalidasa malavikagnimitra
sanskrit kalidasa raghuvamsha
sanskrit baana kadambari
sanskrit bhasa pratijnayogandhararayana"""
```

**a) Question 1 : find the number of books**

In [3]:
```
# Solution: Count the number of lines in the string.
print("# of books : ", len(all.split('\n')))
```

```
# of books :  13
```

In [6]:
```
# enumerate the # of books
print("# of books : ", len(all.split('\n')))
# Enumerate() method adds a counter to an iterable and
#  returns it in a form of enumerate object.
for l in enumerate(all.split('\n')):
    print(l)
```

```
# of books :  13
(0, 'sanskrit kalidasa shakuntala')
(1, 'english r_k_narayan malgudi_days')
(2, 'kannada kuvempu ramayanadarshanam')
(3, 'sanskrit bhasa swapnavasavadatta')
(4, 'kannada kuvempu malegalalli_madumagalu')
(5, 'english r_k_narayan dateless_diary')
(6, 'kannada karanta chomanadudi')
(7, 'sanskrit baana harshacharita')
(8, 'kannada karanta sarasatammana_Samadhi')
(9, 'sanskrit kalidasa malavikagnimitra')
(10, 'sanskrit kalidasa raghuvamsha')
(11, 'sanskrit baana kadambari')
(12, 'sanskrit bhasa pratijnayogandhararayana')
```

**b) Question 2: find the number of languages**

Solution:

- Split the string into lines
- Split each line and extract the first entry only
- Put them into a data structure where the entries shall be unique – that is set.
- Count them.

In [12]:
```
langset = set()
for line in all.split('\n'):
    linewords= line.split()
    langset.add(linewords[0])
print("# of lang : ", len(langset))
```

```
# of lang :  3
```

In [13]:
```
langset = set()
for line in all.split('\n'):
    langset.add(line.split()[0])
print("# of lang : ", len(langset))
```

```
# of lang :  3
```

#### c) Question 3: count the number of books in each language

This is similar to the last example. But set will not be sufficient. For each language we require a count. We require a language:count pair where the languages are unique. The data structure for this is dict.

In these sorts of problems where a data structure has to be created, we will initialize before a loop. We build the data structure element by element within the loop. If the lang is not present, we create the lang as the key and make the corresponding value 0 in the dict. We then count that book by adding one to the value stored in the value field for that language.

In [5]:
```
lang_book_count = {}     # create an empty dict
for line in all.split('\n'):
    lang = line.split()[0]
    if lang not in lang_book_count :
        lang_book_count[lang] = 0
    lang_book_count[lang] += 1

for lang in lang_book_count:
    print(lang, " => ", lang_book_count[lang])
```

```
sanskrit  =>  7
english  =>  2
kannada  =>  4
```

#### d) Question 4: find list of authors for each language

Names of the authors would repeat as each author may have more than one book. So the data structure shall be a dict where key is the language and the value is a set of authors. Check the comments at the end of each line.

In [9]:
```
lang_author = {} # create an empty dict
for line in all.split('\n'):
    (lang, author) = line.split()[:2] # slice and pick up the first two elements
    if lang not in lang_author:       # if key lang does not exist, add that key
        lang_author[lang] = set()     # make the value an empty set
    lang_author[lang].add(author)     # add to the set uniquely
print(lang_author)
```

```
{'sanskrit': {'kalidasa', 'bhasa', 'baana'}, 'english': {'r_k_narayan'}, 'kannada': {'ku
vempu', 'karanta'}}
```

#### Find list of authors for each language

In [10]:
```
for lang in lang_author:
    print(lang)
    for author in lang_author[lang]:
        print("\t", author)
```

```
sanskrit
	 kalidasa
	 bhasa
	 baana
english
	 r_k_narayan
kannada
	 kuvempu
	 karanta
```

#### e) Question 5: find the number of books of each author in each language.

- The data structure shall be a dict of dict of int.
- The key for the outer dict shall be the lang.

- The key for the inner dict will be the author.
- The value shall be int. Check the comments at the end of each line.

In [13]:
```
lang_author = {} # empty dict
for line in all.split('\n'):
    (lang, author) = line.split()[:2]    # slice and pick up what is required
    if lang not in lang_author :                 # check and create an empty dict as the
        lang_author[lang] = {}
    if author not in lang_author[lang] : # if the key does not exist, put the key
        lang_author[lang][author] = 0   # with the value as 0
    lang_author[lang][author] += 1        # increment the count

for lang in lang_author:
    print(lang)
    for author in lang_author[lang]:
        print("\t", author, "=>",
        lang_author[lang][author])
```

```
sanskrit
         kalidasa => 3
         bhasa => 2
         baana => 2
english
         r_k_narayan => 2
kannada
         kuvempu => 2
         karanta => 2
```

**f) Question 6: create a language to author to book mapping.**

We will create a dict as the solution. The key will be the language. The value will be a dict. In that dict, key will be the author and the value will be a list.

In [15]:
```
info = {} # create an empty dict
for line in all.split('\n'):
    (lang, author, title) = line.split() # get the required info by splitting
    if lang not in info :   # if lang does not exist, add that as the key in info
        info[lang] = {}
    if author not in info[lang] :    # if author does not exist, add that as the key in
        info[lang][author] = []    # make the value an empty list
    info[lang][author].append(title) # append to the list.

for lang in info:
    print(lang)
    for author in info[lang]:
        print("\t", author)
        for title in info[lang][author]:
            print("\t\t", title)
```

```
sanskrit
         kalidasa
                 shakuntala
                 malavikagnimitra
                 raghuvamsha
         bhasa
                 swapnavasavadatta
                 pratijnayogandhararayana
         baana
                 harshacharita
                 kadambari
english
         r_k_narayan
                 malgudi_days
                 dateless_diary
```

```
kannada
        kuvempu
                ramayanadarshanam
                malegalalli_madumagalu
        karanta
                chomanadudi
                sarasatammana_Samadhi
```

## The Big-O notation and its importance in algorithm selection

Big-O notation describes the performance of algorithms concerning their input size. As a senior Python developer, you should be comfortable working with Big-O notation to choose the right algorithm for your tasks.

Using an algorithm with a lower Big-O complexity can lead to huge performance improvements.

For example, consider these two functions that count the occurrences of elements in a list:

In [2]:
```python
def count_elements_slow(input_list):
    output = {}
    for item in input_list:
        output[item] = input_list.count(item)
    return output

def count_elements_fast(input_list):
    output = {}
    for item in input_list:
        output[item] = output.get(item, 0) + 1
    return output
```

The first function count_elements_slow has a time complexity of O(n^2), while the second function count_elements_fast has a time complexity of O(n). Therefore, the second function will perform much faster for large input lists.

## Code Optimization Techniques

Writing efficient code is an art and a science. With some simple tricks and techniques, you can optimize the performance of your Python code significantly.

### Loop optimization and avoiding nested loops

Loop optimization is a crucial aspect of code optimization. To speed up loops in Python, consider the following tips:

Use built-in functions like map, filter, and zip instead of loops whenever possible Minimize work done inside the loop (e.g., move function calls and calculations outside the loop if they don't depend on loop variables) Avoid using nested loops when possible, as they significantly increase complexity

In [3]:
```python
# Simple loop optimization example
input_list = [1, 2, 3, 4, 5]

# Slow loop
result_slow = []
for item in input_list:
    squared = item**2
    result_slow.append(squared)
```

```
# Optimized loop using a list comprehension
result_fast = [item**2 for item in input_list]
```

Warning: Use Yellow for a warning that might need attention.

References:


1. set_dict.pdf – Prof. N S Kumar, Dept. of CSE, PES University.
2. https://www.w3schools.com/python
3. https://docs.python.org/