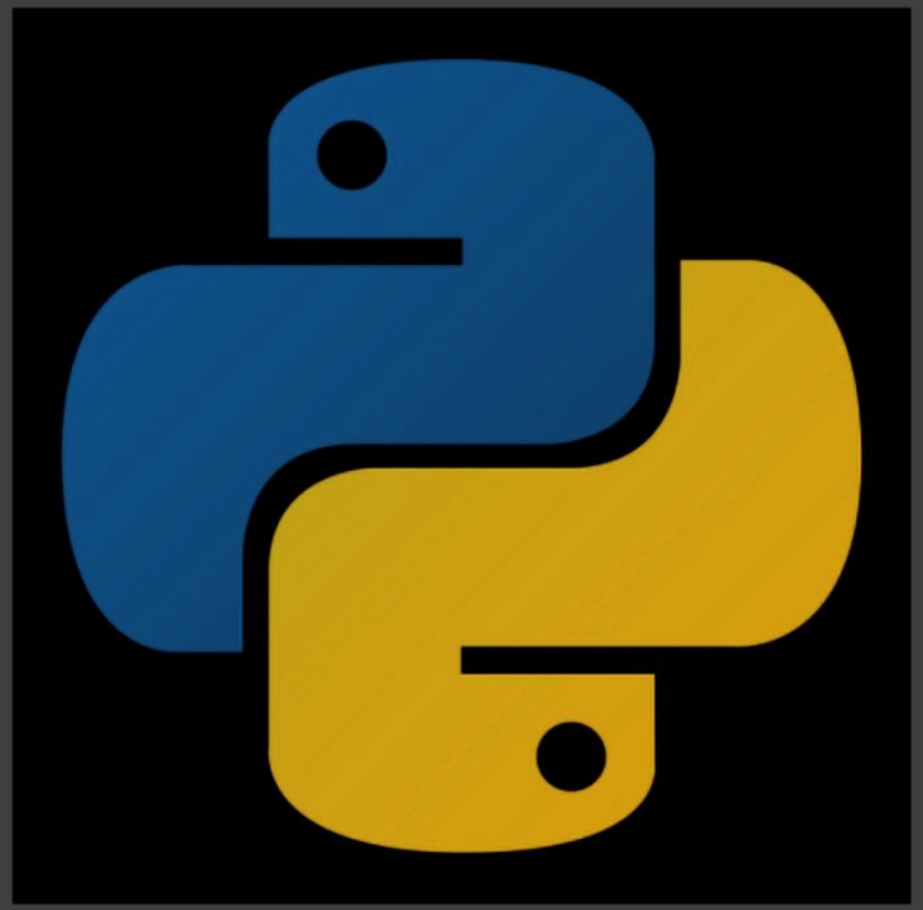


Python for Beginners

Glimpse of last lecture

Introduction (an overview to course)
- Getting started "Hello World"
- Keywords and identifiers
- Comments and Statements
- Variables and assignments
- Data types
- Flow control
- Methods and Functions
- Reading and Writing files
- Modules and import
- Object Oriented Programming

Data types
- Numbers
- List
- Tuple
- String
- Dictionary
- Set

Program Flow Control in Python
- *If* statement
- Elif statement
- More on if, elif and else
- *For* loop
- *While* loop
- Useful operators

Methods and Functions
- Defining a function
- Flow when calling a function
- Parameters and arguments
- Global/local
- Functions calling functions

Reading/Writing Files
- Files and directories in Python
- Reading from a file
- Parsing data
- Printing data to external file

Modules and Import
- Standard Python library
- Datetime module
- Math and Random module
- Generators and decorators
- NumPy, Pandas, Matplotlib (basic uses)

Object Oriented Programming
- Introduction to OOPs
- Attributes and Class keywords
- Inheritance and Polymorphism

Statistical analysis of data with Python

Glimpse of last lecture

Introduction (an overview to course)
- Getting started "Hello World"
- Keywords and identifiers
- Comments and Statements
- Variables and assignments
- Data types
- Flow control
- Methods and Functions
- Reading and Writing files
- Modules and import
- Object Oriented Programming

Data types
- Numbers
- List
- Tuple
- String
- Dictionary
- Set

Program Flow Control in Python
- *If* statement
- Elif statement
- More on if, elif and else
- *For* loop
- *While* loop
- Useful operators

Methods and Functions
- Defining a function
- Flow when calling a function
- Parameters and arguments
- Global/local
- Functions calling functions

Reading/Writing Files
- Files and directories in Python
- Reading from a file
- Parsing data
- Printing data to external file

Modules and Import
- Standard Python library
- Datetime module
- Math and Random module
- Generators and decorators
- NumPy, Pandas, Matplotlib (basic uses)

Object Oriented Programming
- Introduction to OOPs
- Attributes and Class keywords
- Inheritance and Polymorphism

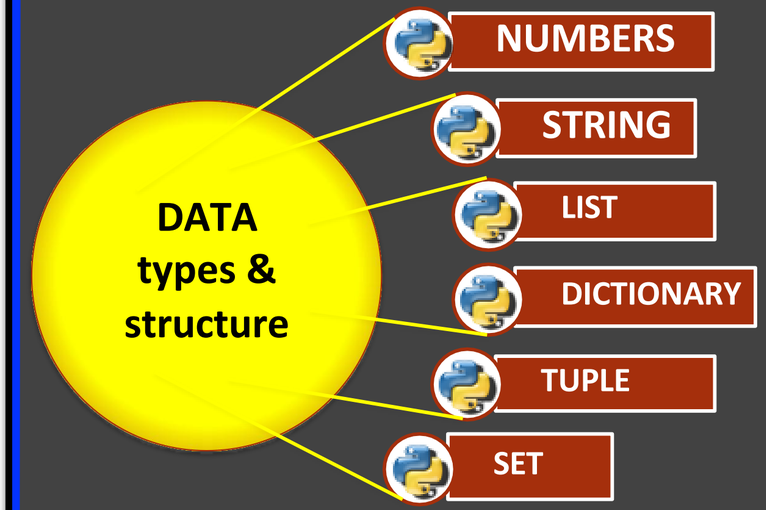
Statistical analysis of data with Python

Variables: are used to *hold data* during execution the program.

Identifiers: is a name, to identify *variable*, *function*, *module*….

Keywords: predefined words, that has specific meaning and can't be used as constants, variables or other identifier names.

Glimpse of last lecture



SEQUENCE type

- **NUMBERS** → **Numeric values** (e.g., Integers, real numbers, complex)
- **STRING** → **Contiguous set** of ***characters*** in quotation marks (‘Hello’)
- **LIST** → **Compound data type** : composed of *items* in square bracket[], separated by commas [‘abc’, 5, 10.5, ‘xyz’]
- **TUPLE** → **Similar to the list**, however, tuples are **enclosed within parenthesis** (‘abc’, 786, 2.23, ‘john’)
- **DICTIONARY** → **Dictionaries** consist of key-value pairs {1001: "John", 1002: "Jane”}
- **SET** →
- **Boolean** → True (T) or False (F)

Variable : are used to ***hold data*** during execution the program

Variable name:

There are certain rules, one has to take care:

- Names can’t start with a number (they can be alphanumeric)
- No space in the name of variable, use “_” instead (if required)
- Following symbols can’t be used.
  :,””, <>,/,?,|,\,(),!,@,&,%,…..

Variable : are used to ***hold data*** during execution the program

Variable name:

There are certain rules, one has to take care:

- Names can’t start with a number (they can be alphanumeric)
- No space in the name of variable, use “_” instead (if required)
- Following symbols can’t be used.
  :,””, <>,/,?,|,\,(),!,@,&,%,…..

Avoid using words (keywords), that have special meaning:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

Link : https://docs.python.org/3/reference/lexical_analysis.html#keywords

Variable : are used to ***hold data*** during execution the program

Variable name:

There are certain rules, one has to take care:

- Names can't start with a number (they can be alphanumeric)
- No space in the name of variable, use "_" instead (if required)
- Following symbols can't be used.
  :,"", <>,/,?,|,\,(),!,@,&,%,…..

Avoid using words (keywords), that have special meaning:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

Link : https://docs.python.org/3/reference/lexical_analysis.html#keywords

How to know, if a variable is "keyword" identifier:

syntax highlighting

OR

```
Import keyword
keyword.iskeyword( '?' )
```

Print Keyword list:

```
import keyword
keyword.kwlist
```

Dynamic typing - feature or flaw

Python is very flexible as concerns the data type ,

it offers the Dynamic typing, this means,

one can reassign the variable to different data type.

```
city_schools = 2

city_schools = [“private”, “public”]
```

Okay in Python, but will show errors in other programming language which are statically typed

Dynamic typing

- **Pros** Easy to work with and Faster development
- **Cons** May lead to bug for unexpected data type

```
type(city_schools)

type(10.55)
```

Dynamic typing - feature or flaw

Python is very flexible as concerns the data type ,

it offers the Dynamic typing, this means,

one can reassign the variable to different data type.

```
city_schools = 2

city_schools = [“private”, “public”]
```

Okay in Python, but will show errors in other programming language which are statically typed

Dynamic typing

- **Pros** Easy to work with and Faster development
- **Cons** May lead to bug for unexpected data type

```
type(city_schools)

type(10.55)
```

Dynamic typing - feature or flaw

Python is very flexible as concerns the data type ,

it offers the Dynamic typing, this means,

one can reassign the variable to different data type.

```
city_schools = 2

city_schools = [“private”, “public”]
```

Okay in Python, but will show errors in other programming language which are statically typed

Dynamic typing

- **Pros** Easy to work with and Faster development
- **Cons** May lead to bug for unexpected data type

```
type(city_schools)

type(10.55)
```

Feature, but use with precaution

Strings

Strings in python are mainly used to *store text information in a sequence*. This allows one to access the individual element of the string using the corresponding *index number*. In python, indexing starts with **0**

Name = 'JAGIELLONIAN UNIVERSITY'

	J	A	G	I	E	L	L	O	N	I	A	N		U	N	I	V	E	R	S	I	T	Y
Index number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

In python, one can use *single/double quote for strings*. One has to take precaution in some cases, for e.g.,

' I'm not responsible for the computers in lab '

" I'm not responsible for the computers in lab "

We have already learned to create and print the strings. Today, we will practice the ***indexing and slicing*** of strings.

Strings

Strings in python are mainly used to *store text information in a sequence*. This allows one to access the individual element of the string using the corresponding *index number*. In python, indexing starts with **0**

Name = 'JAGIELLONIAN UNIVERSITY'

	J	A	G	I	E	L	L	O	N	I	A	N		U	N	I	V	E	R	S	I	T	Y
Index number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

In python, one can use *single/double quote for strings*. One has to take precaution in some cases, for e.g.,

' I'm not responsible for the computers in lab '

" I'm not responsible for the computers in lab "

We have already learned to create and print the strings. Today, we will practice the ***indexing and slicing*** of strings.

Strings

Strings in python are mainly used to *store text information in a sequence*. This allows one to access the individual element of the string using the corresponding *index number*. In python, indexing starts with **0**

Name = 'JAGIELLONIAN UNIVERSITY'

	J	A	G	I	E	L	L	O	N	I	A	N		U	N	I	V	E	R	S	I	T	Y
Index number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

In python, one can use *single/double quote for strings*. One has to take precaution in some cases, for e.g.,

' I'm not responsible for the computers in lab '

" I'm not responsible for the computers in lab "

We have already learned to create and print the strings. We will practice the

***indexing and* slicing** of strings.

Lets workout with the strings

name = “JAGIELLONIAN UNIVERSITY”

Calculate the length of the string using built-in function

Access, a particular element from the string name

Print the selected part/slicing of the string

Strings are ***immutable***

Concatenation allowed !!!

```
# Calculating length of a string, use print(len(str))

# Indexing to access elements of string
    print(name[i])

# Slicing of string
print(name[3:])   name[:3]   name[-2]   name[-2:]
name[::1]   name[::3]   #Everything but in
                            step size/s

# Immutability
name[0] = ‘L’   #Should throw an error,
                    no item assignment in str

# Concatenation
Name =
name + ‘ is the oldest university in KRAKOW’
```

“ Formatting Python string ”

Using .format method

```
name_course = “This is {initial} course for beginners”.format(initial=‘Python’)

name_course
```

.format( ) method comes with a lot of variation to format and print output, we will learn detailed potentialities in next lectures

Using *formatting operator* % (*placeholders*)

```
name_course = “This is %s course for beginners”%'Python'
```

```
name_course = “This is %r course for beginners”%'Python'
```

“ Formatting numbers ”

Using % (*placeholders*)

```
Marks = “In Graduation I got %5.2f percent marks but in masters I got %6.3f percent marks:”%(73.778,84.54432)
```

Hands-On

```
Text1 = “I like programming when there is someone to code for me”
Text2 = “I like {2} when there is {1} to code for me”.format(‘programming’,’someone’)
Total_sum = “Spending %.f on mangoes and %5.3f on banana cost me”%(70.5566,345.34456)
```

“ Formatting Python string ”

Using .format method

```
name_course = “This is {initial} course for beginners”.format(initial=‘Python’)

name_course
```

.format( ) method comes with a lot of variation to format and print output, we will learn detailed potentialities in next lectures

Using *formatting operator* % (*placeholders*)

```
name_course = “This is %s course for beginners”%'Python'
```

```
name_course = “This is %r course for beginners”%'Python'
```

“ Formatting numbers ”

Using % (*placeholders*)

```
Marks = “In Graduation I got %5.2f percent marks but in masters I got %6.3f percent marks:”%(73.778,84.54432)
```

Hands-On

```
Text1 = “I like programming when there is someone to code for me”
Text2 = “I like {1} when there is {0} to code for me”.format(‘programming’,’someone’)
Total_sum = “Spending %.f on mangoes and %5.3f on banana cost me”%(70.5566,345.34456)
```

Useful built-in String functions/methods

Python facilitates with several built-in methods which can be accessed, to execute actions:

Here we ***discuss*** few of the **most frequently** used on Strings

name = “JAGIELLONIAN UNIVERSITY”

Useful built-in String functions/methods

Python facilitates with several built-in methods which can be accessed, to execute actions:

Here we ***discuss*** few of the **most frequently** used on Strings

name = "JAGIELLONIAN UNIVERSITY"

split a string by blank space # default

split a string by a specific element (# element not included)

name.split( ) name.split('A')

Write string in upper / lower case

name.upper( ) name.lower( )

find commands for the strings

name.find("T") name.find("T", 0, 11)

.format( ) ; format the specified value and insert them in string placeholder defined

```
name_university = "My university name is {initial} University".format(initial='Jagiellonian')
name_university
```

Replace : replace a specified value inside the string ; Replace J with Y

name.replace('J','Y')

Useful built-in String functions/methods

capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found

rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isascii()	Returns True if all characters in the string are ascii characters
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Converts the elements of an iterable into a string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found

LISTS - [ ]

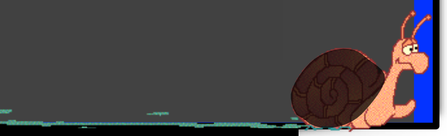
Lists is another *sequence data type* and in comparison to strings,

"lists are ***mutable*** and can be considered more general *version of sequence data type*"

- List can be constructed as group of elements, separated by commas inside the square brackets [ ]
- Lists can be *constructed of strings, numbers and even with possibility of nested list*
- Several methods, which were used on String, can also be implement on LIST
- List elements can also be accessed by using the *index number*.

We will learn briefly about

- creating lists,
- indexing and slicing like in case of strings,
- go through basic in-built methods to operate on lists, and finally
- nested listing



LISTS - [ ]

Lists is another *sequence data type* and in comparison to strings,

"lists are ***mutable*** and can be considered more general *version of sequence data type*"

- List can be constructed as group of elements, separated by commas inside the **square brackets [ ]**
- Lists can be ***constructed of strings***, ***numbers*** *and* ***even*** *with possibility of* ***nested list***
- Several methods, which were used on String, can also be implement on LIST
- List elements can also be accessed by using the ***index number***.

We will learn briefly about

- creating lists,
- indexing and slicing like in case of strings,
- go through basic in-built methods to operate on lists, and finally
- nested listing

LISTS - [ ]

Lists is another *sequence data type* and in comparison to strings,

"lists are ***mutable*** and can be considered more general *version of sequence data type*"

- List can be constructed as group of elements, separated by commas inside the **square brackets [ ]**
- Lists can be ***constructed of strings***, ***numbers*** *and* ***even*** *with possibility of* ***nested list***
- Several methods, which were used on String, can also be implement on LIST
- List elements can also be accessed by using the ***index number***.

We will learn briefly about

- creating lists,
- **indexing** and **slicing** like in case of strings,
- go through basic **in-built methods** to operate on lists, and finally
- **nested** listing

LISTS - [ ]

my_first_list = [ ]

my_first_list = [‘Jagiellonian University’, 1364, ‘Bio, Phy, Che’]

Length of the list

```
print(len(my_first_list))
```

```
my_first_list[0]
```

Indexing and slicing of the list

```
my_first_list[1:]
my_first_list[:-1]
```

Concatenation, adding new element

Not change the original list, adding temporarily. For permanent addition ?

```
my_first_list + [‘medial physics’]
```

Use mutable feature, re-assigning element

```
my_first_list =
        my_first_list + [‘medical physics’]
```

Repeat the element of list

```
my_first_list * 5
```

LISTS - [ ]

my_first_list = [ ]

my_first_list = [‘Jagiellonian University’, 1364, ‘Bio, Phy, Che’]

Length of the list

```
print(len(my_first_list))
```

Indexing and **slicing** of the list

```
my_first_list[0]
my_first_list[1:]
my_first_list[:-1]
```

Concatenation, adding new element

Not change the original list, adding temporarily. For permanent addition ?

```
my_first_list + [‘medial physics’]
```

Use mutable feature, re-assigning element

```
my_first_list =
    my_first_list + [‘medical physics’]
```

Repeat the element of list

```
my_first_list * 5
```

LISTS - [ ]

my_first_list = [ ]

my_first_list = [‘Jagiellonian University’, 1364, ‘Bio, Phy, Che’]

Length of the list

```
print(len(my_first_list))
```

Indexing and **slicing** of the list

```
my_first_list[0]
my_first_list[1:]
my_first_list[:-1]
```

Concatenation of list, adding new element

Not change the original list, adding temporarily, For permanent addition ?

```
my_first_list + [‘medial physics’]
```

Use mutable feature, re-assigning element

```
my_first_list =
    my_first_list + [‘medical physics’]
```

Repeat the element of list

```
my_first_list * 5
```

LISTS - [ ]

my_first_list = [ ]

my_first_list = [‘Jagiellonian University’, 1364, ‘Bio, Phy, Che’]

Length of the list

```
print(len(my_first_list))
```

Indexing and **slicing** of the list

```
my_first_list[0]
my_first_list[1:]
my_first_list[:-1]
```

Concatenation of list, adding new element

Not change the original list, adding temporarily, For permanent addition ?

```
my_first_list + [‘medial physics’]
```

Use **mutable feature**, re-assigning element

```
my_first_list =
my_first_list + [‘medical physics’]
```

Repeat the element of list

```
my_first_list * 5
```

LISTS - [ ]

my_first_list = [ ]

my_first_list = [‘Jagiellonian University’, 1364, ‘Bio, Phy, Che’]

Length of the list

```
print(len(my_first_list))
```

Indexing and **slicing** of the list

```
my_first_list[0]
my_first_list[1:]
my_first_list[:-1]
```

Concatenation of list, adding new element

Not change the original list, adding temporarily, For permanent addition ?

```
my_first_list + [‘medial physics’]
```

Use **mutable feature**, re-assigning element

```
my_first_list =
    my_first_list + [‘medical physics’]
```

Repeat the element of list

```
my_first_list * 5
```

Most frequently used methods with LISTS - [ ]

Method name	Description
sort( )	**Sorts the elements in the list**
reverse( )	**Reverses the order of the list**
index( )	**Returns the index of the element with the specified value**
extend( )	Add the *elements of a list*, *to the end of the current list*
clear( )	**Removes all the elements from the list**
insert( )	Adds an element at the specified position (**index,element**)
count( )	**Returns the number of times elements appear in list**
copy( )	**Returns a copy of the list**
remove( )	**Removes the item with the specified value**
pop( )	**Removes the element at the specified position**
append( )	Adds an element at the end of the list

```
# Sample list
my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

my_list.sort()
print("Sorted list:", my_list)

my_list.reverse()
print("Reversed list:", my_list)

index_of_5 = my_list.index(5)
print("Index of first occurrence of 5:", index_of_5)

my_list.extend([7, 8, 9])
print("List after extending:", my_list)

my_list.clear()
print("List after clearing:", my_list)
```

Most frequently used methods with LISTS - [ ]

Method name	Description	# Sample list
		my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

sort( )	**Sorts the elements in the list**	**my_list.sort() print("Sorted list:", my_list)**
reverse( )	**Reverses the order of the list**	**my_list.reverse() print("Reversed list:", my_list)**
index( )	**Returns the index of the element with the specified value**	**index_of_5 = my_list.index(5) print("Index of first occurrence of 5:", index_of_5)**
extend( )	Add the *elements of a list*, *to the end of the current list*	**my_list.extend([7, 8, 9]) print("List after extending:", my_list)**
clear( )	**Removes all the elements from the list**	**my_list.clear() print("List after clearing:", my_list)**
insert( )	Adds an element at the specified position (**index,element**)	
count( )	**Returns the number of times elements appear in list**	
copy( )	**Returns a copy of the list**	
remove( )	**Removes the item with the specified value**	
pop( )	**Removes the element at the specified position**	
append( )	Adds an element at the end of the list	

Most frequently used methods with LISTS - [ ]

Method name	Description
sort( )	**Sorts the elements in the list**
reverse( )	**Reverses the order of the list**
index( )	**Returns the index of the element with the specified value**
extend( )	Add the *elements of a list*, *to the end of the current list*
clear( )	**Removes all the elements from the list**
insert( )	Adds an element at the specified position (**index,element**)
count( )	**Returns the number of times elements appear in list**
copy( )	**Returns a copy of the list**
remove( )	**Removes the item with the specified value**
pop( )	**Removes the element at the specified position**
append( )	Adds an element at the end of the list

 # Reinitialize the list for further examples

my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

Most frequently used methods with LISTS - [ ]

Method name	Description	
sort( )	**Sorts the elements in the list**	
reverse( )	**Reverses the order of the list**	
index( )	**Returns the index of the element with the specified value**	
extend( )	Add the *elements of a list*, *to the end of the current list*	my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
clear( )	**Removes all the elements from the list**	
insert( )	Adds an element at the specified position (**index,element**)	**my_list.insert(2, 7)** **print("List after inserting 7 at index 2:", my_list)**
count( )	**Returns the number of times elements appear in list**	**count_of_1 = my_list.count(1)** **print("Count of 1:", count_of_1)**
copy( )	**Returns a copy of the list**	**my_list_copy = my_list.copy()** **print("Copied list:", my_list_copy)**
remove( )	**Removes the item with the specified value**	**my_list.remove(7)** **print("List after removing first occurrence of 7:", my_list)**
pop( )	**Removes the element at the specified position**	**popped_element = my_list.pop(3)** **print("Popped elem. index 3:", popped_element)** **print("List after popping:", my_list)**
append( )	Adds an element at the end of the list	**my_list.append(10)** **print("List after appending 10:", my_list)**

Tuples - ( )

- **Tuples** are similar to LISTS, However, tuples are 'immutable',
  "thus are preferable when data integrity is required"

- **Tuples** use parentheses instead of square brackets (like in list)

```
my_first_tuple = ('apple', 'mango', 'banana')
```

- **Tuples** are **Ordered** ; indexing allowed.

- **Tuples** are **Unchangeable (immutable)** ; once create, can't edit (change, add or remove)

- **Tuples** allows duplicity

```
my_first_tuple = ('apple', 'mango', 'banana', 'mango', 'apple')
```

- **Tuples** can contain different data types

```
my_first_tuple = ('apple', 34, True, 'mango')
```

- **Tuples** can't be made of one element

```
my_first_tuple = ('apple',)
```

comma allows to construct such one item tuple

- ***Access tuple element using indexing (using index( ));***
  Calculate the length of tuple (using len( )) ;
  Returns number of times a specified value/element occurs (using count( ))

Hands-On tuple

Create tuple : `fruits_tuple = ('apple', 'banana', 'mango')`

Ordering (indexing / slicing)

```
print("First element:", fruits_tuple[0])
print("Last element:", fruits_tuple[-1])
```

Tuple duplicates :

```
duplicate_tuple = ('apple', 'mango', 'banana', 'mango', 'apple')
print("Tuple with duplicates:", duplicate_tuple)
```

Tuple duplicates :

```
fruits_tuple[1] = 'orange'
```

Find length of tuple :

```
print("Length of duplicate_tuple:", len(duplicate_tuple))
```

Find index of an element in tuple :

```
banana_index = duplicate_tuple.index('banana')
print("Index of 'banana':", banana_index)
```

How many times an element appears in tuple :

```
apple_count = duplicate_tuple.count('apple')
print("Number of times 'apple' appears:", apple_count)
```

Tuple concatenation :

```
Final_tuple = fruits_tuple + duplicate_tuple
print(final_tuple)
```

Tuple Unpacking :

```
my_tuple = (10, 20, 30)
a, b, c = my_tuple
print(a, b, c)
```

Membership Test :

```
my_tuple = (10, 20, 30)
print(20 in my_tuple)
print(40 not in my_tuple)
```

Tuple repetition :

```
my_tuple = (1, 2)
print(my_tuple * 3)
```

Built-in Functions (min() and max())

```
my_tuple = (10, 20, 30)
print(min(my_tuple))
print(max(my_tuple))
```

Hands-On tuple

Create tuple : `fruits_tuple = ('apple', 'banana', 'mango')`

Ordering (indexing / slicing)

```
print("First element:", fruits_tuple[0])
print("Last element:", fruits_tuple[-1])
```

Create tuple duplicates :

```
duplicate_tuple = ('apple', 'mango', 'banana', 'mango', 'apple')
print("Tuple with duplicates:", duplicate_tuple)
```

Tuple duplicates :

```
fruits_tuple[1] = 'orange'
```

Find length of tuple :

```
print("Length of duplicate tuple:", len(duplicate_tuple))
```

Find index of an element in tuple :

```
banana_index = duplicate_tuple.index('banana')
print("Index of 'banana':", banana_index)
```

How many times an element appears in tuple :

```
apple_count = duplicate_tuple.count('apple')
print("Number of times 'apple' appears:", apple_count)
```

Tuple concatenation :

```
Final_tuple = fruits_tuple + duplicate_tuple
print(final_tuple)
```

Tuple Unpacking :

```
my_tuple = (10, 20, 30)
a, b, c = my_tuple
print(a, b, c)
```

Membership Test :

```
my_tuple = (10, 20, 30)
print(20 in my_tuple)
print(40 not in my_tuple)
```

Tuple repetition :

```
my_tuple = (1, 2)
print(my_tuple * 3)
```

Built-in Functions (min() and max())

```
my_tuple = (10, 20, 30)
print(min(my_tuple))
print(max(my_tuple))
```

Hands-On tuple

Create tuple : `fruits_tuple = ('apple', 'banana', 'mango')`

Ordering (indexing / slicing)

```
print("First element:", fruits_tuple[0])
print("Last element:", fruits_tuple[-1])
```

Create tuple duplicates :

```
duplicate_tuple = ('apple', 'mango', 'banana', 'mango', 'apple')
print("Tuple with duplicates:", duplicate_tuple)
```

Tuple duplicates :

```
fruits_tuple[1] = 'orange'
```

Find length of tuple :

```
print("Length of duplicate tuple:", len(duplicate_tuple))
```

Find index of an element in tuple :

```
banana_index = duplicate_tuple.index('banana')
print("Index of 'banana':", banana_index)
```

How many times an element appears in tuple :

```
apple_count = duplicate_tuple.count('apple')
print("Number of times 'apple' appears:", apple_count)
```

Tuple concatenation :

```
Final_tuple = fruits_tuple + duplicate_tuple
print(final_tuple)
```

Tuple Unpacking :

```
my_tuple = (10, 20, 30)
a, b, c = my_tuple
print(a, b, c)
```

Membership Test :

```
my_tuple = (10, 20, 30)
print(20 in my_tuple)
print(40 not in my_tuple)
```

Tuple repetition :

```
my_tuple = (1, 2)
print(my_tuple * 3)
```

Built-in Functions (min() and max())

```
my_tuple = (10, 20, 30)
print(min(my_tuple))
print(max(my_tuple))
```

Hands-On tuple

Create tuple : `fruits_tuple = ('apple', 'banana', 'mango')`

Ordering (indexing / slicing)

```
print("First element:", fruits_tuple[0])
print("Last element:", fruits_tuple[-1])
```

Create tuple duplicates :

```
duplicate_tuple = ('apple', 'mango', 'banana', 'mango', 'apple')
print("Tuple with duplicates:", duplicate_tuple)
```

Tuple duplicates :

```
fruits_tuple[1] = 'orange'
```

Find length of tuple :

```
print("Length of duplicate tuple:", len(duplicate_tuple))
```

Find index of an element in tuple :

```
banana_index = duplicate_tuple.index('banana')
print("Index of 'banana':", banana_index)
```

How many times an element appears in tuple :

```
apple_count = duplicate_tuple.count('apple')
print("Number of times 'apple' appears:", apple_count)
```

Tuple concatenation :

```
Final_tuple = fruits_tuple + duplicate_tuple
print(final_tuple)
```

Tuple Unpacking :

```
my_tuple = (10, 20, 30)
a, b, c = my_tuple
print(a, b, c)
```

Membership Test :

```
my_tuple = (10, 20, 30)
print(20 in my_tuple)
print(40 not in my_tuple)
```

Tuple repetition :

```
my_tuple = (1, 2)
print(my_tuple * 3)
```

Built-in Functions (min() and max())

```
my_tuple = (10, 20, 30)
print(min(my_tuple))
print(max(my_tuple))
```

Hands-On tuple

Create tuple : `fruits_tuple = ('apple', 'banana', 'mango')`

Ordering (indexing / slicing)

```
print("First element:", fruits_tuple[0])
print("Last element:", fruits_tuple[-1])
```

Create tuple duplicates :

```
duplicate_tuple = ('apple', 'mango', 'banana', 'mango', 'apple')
print("Tuple with duplicates:", duplicate_tuple)
```

Tuple duplicates :

```
fruits_tuple[1] = 'orange'
```

Find length of tuple :

```
print("Length of duplicate tuple:", len(duplicate_tuple))
```

Find index of an element in tuple :

```
banana_index = duplicate_tuple.index('banana')
print("Index of 'banana':", banana_index)
```

How many times an element appears in tuple :

```
apple_count = duplicate_tuple.count('apple')
print("Number of times 'apple' appears:", apple_count)
```

Tuple concatenation :

```
Final_tuple = fruits_tuple + duplicate_tuple
print(final_tuple)
```

Tuple Unpacking :

```
my_tuple = (10, 20, 30)
a, b, c = my_tuple
print(a, b, c)
```

Membership Test :

```
my_tuple = (10, 20, 30)
print(20 in my_tuple)
print(40 not in my_tuple)
```

Tuple repetition :

```
my_tuple = (1, 2)
print(my_tuple * 3)
```

Built-in Functions (min() and max())

```
my_tuple = (10, 20, 30)
print(min(my_tuple))
print(max(my_tuple))
```

Hands-On tuple

Create tuple :
```
fruits_tuple = ('apple', 'banana', 'mango')
```

Ordering (indexing / slicing)
```
print("First element:", fruits_tuple[0])
print("Last element:", fruits_tuple[-1])
```

Create tuple duplicates :
```
duplicate_tuple = ('apple', 'mango', 'banana', 'mango', 'apple')
print("Tuple with duplicates:", duplicate_tuple)
```

Tuple duplicates :
```
fruits_tuple[1] = 'orange'
```

Find length of tuple :
```
print("Length of duplicate tuple:", len(duplicate_tuple))
```

Find index of an element in tuple :
```
banana_index = duplicate_tuple.index('banana')
print("Index of 'banana':", banana_index)
```

How many times an element appears in tuple :
```
apple_count = duplicate_tuple.count('apple')
print("Number of times 'apple' appears:", apple_count)
```

Tuple concatenation :
```
Final_tuple = fruits_tuple + duplicate_tuple
print(final_tuple)
```

Tuple Unpacking :
```
my_tuple = (10, 20, 30)
a, b, c = my_tuple
print(a, b, c)
```

Membership Test :
```
my_tuple = (10, 20, 30)
print(20 in my_tuple)
print(40 not in my_tuple)
```

Tuple repetition :
```
my_tuple = (1, 2)
print(my_tuple * 3)
```

Built-in Functions (min() and max())
```
my_tuple = (10, 20, 30)
print(min(my_tuple))
print(max(my_tuple))
```

Dictionary - { }

Dictionary works based on the key:value concept, similar to mapping

keys are unique and are **immutable** data type - **strings**, **numbers** or **tuples**
but **values** can be of any type

Best_City	2022
Best_University	spicy
Best_place	Volleyball
Best_food	Jagiellonian
Best_sport	Krakow
Year	market_square

```
my_first_dict = {
    'Best_City' : 'Krakow',
    'Best_University' : 'Jagiellonain',
    'Best_place' : 'market_square',
    'Best_food' : 'spicy',
    'Best_sport' : 'volleyball',
    'Year' : 2022
}
print(my_fist_dict)
```

One can access the values, using the key

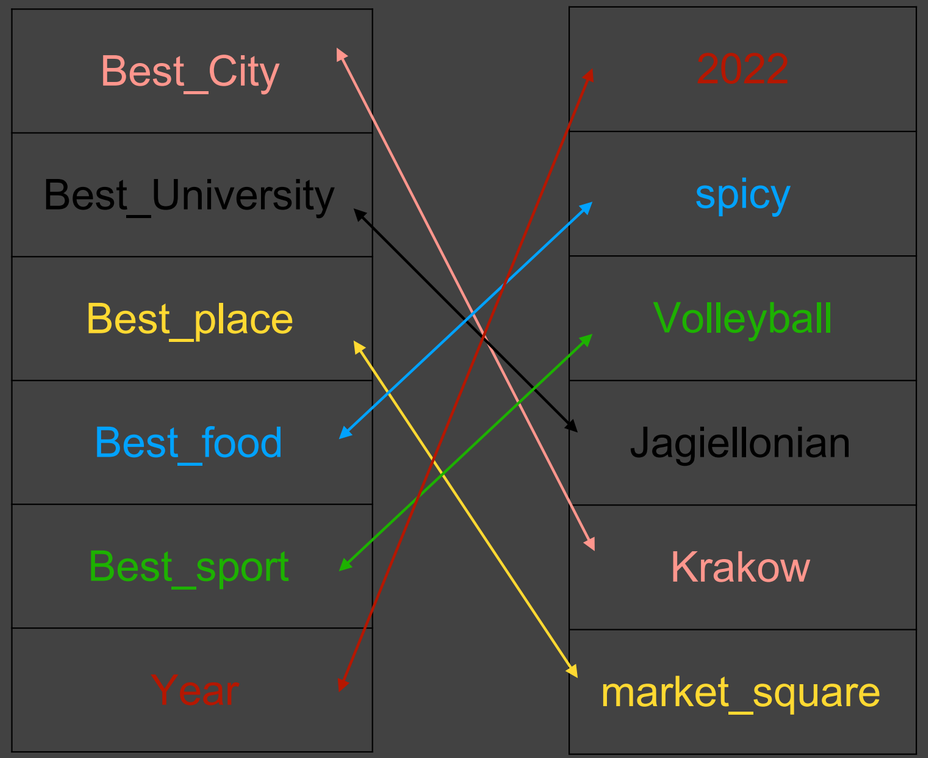
```
print(my_first_dict['Best_place'])
print(my_first_dict['Best_food'])
```

- **After python 3.6, dictionaries are ordered**
- Dictionaries are **changeable** (change, add, remove) after dictionary is created
- Unlike tuple, **dictionaries don't allow duplicates**

Dictionary - { }

Dictionary works based on the key:value concept, similar to mapping

keys are unique and are immutable data type - strings, numbers or tuples
but **values** can be of any type




```
my_first_dict = {
    'Best_City' : 'Krakow',
    'Best_University' : 'Jagiellonain',
    'Best_place' : 'market_square',
    'Best_food' : 'spicy',
    'Best_sport' : 'volleyball',
    'Year' : 2022
}
print(my_fist_dict)
```

One can access the values, using the key

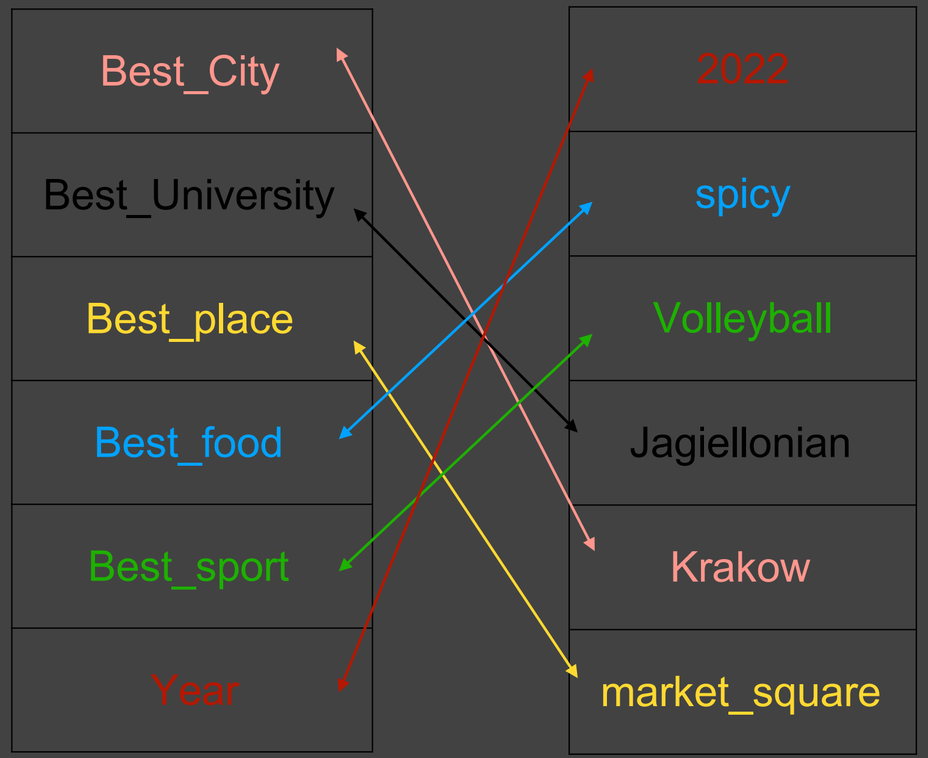
```
print(my_first_dict['Best_place'])
print(my_first_dict['Best_food'])
```

- **After python 3.6, dictionaries are ordered**
- Dictionaries are **changeable** (change, add, remove) after dictionary is created
- Unlike tuple, **dictionaries don't allow duplicates**

Dictionary - { }

Dictionary works based on the key:value concept, similar to mapping

keys are unique and are immutable data type - strings, numbers or tuples
but **values** can be of any type




```
my_first_dict = {
    'Best_City' : 'Krakow',
    'Best_University' : 'Jagiellonain',
    'Best_place' : 'market_square',
    'Best_food' : 'spicy',
    'Best_sport' : 'volleyball',
    'Year' : 2022
}
print(my_fist_dict)
```

One can access the values, using the key

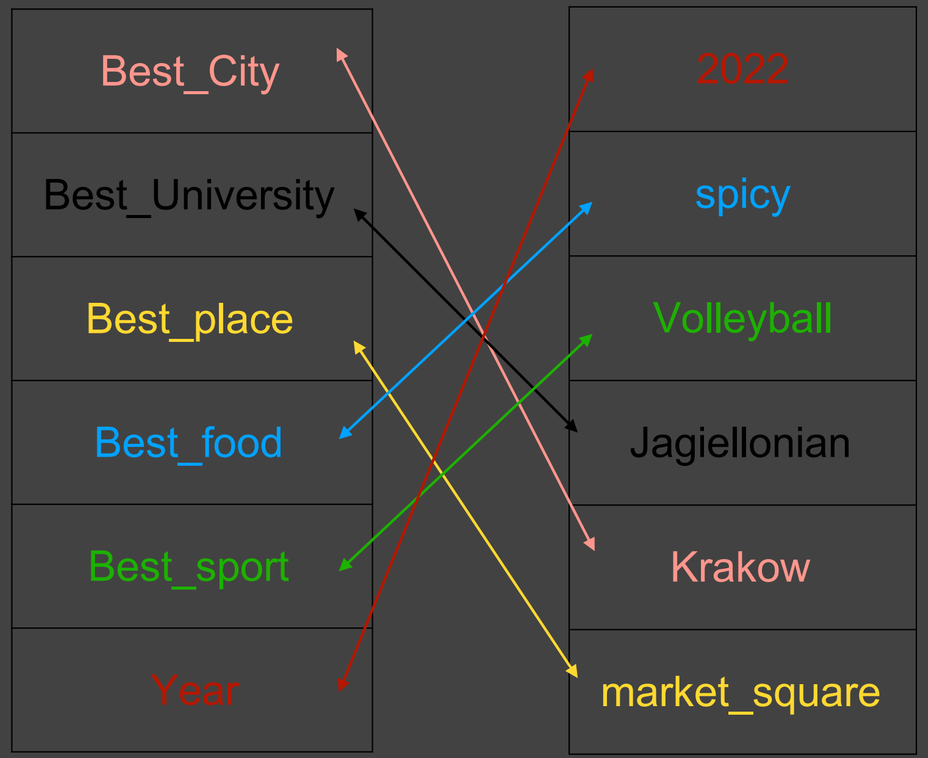
```
print(my_first_dict['Best_place'])
print(my_first_dict['Best_food'])
```

- **After python 3.6, dictionaries are ordered**
- Dictionaries are **changeable** (change, add, remove) after dictionary is created.
- Unlike tuple, **dictionaries don't allow duplicates**

Dictionary - { }

Dictionary works based on the key:value concept, similar to mapping

keys are unique and are immutable data type - strings, numbers or tuples
but **values** can be of any type




```
my_first_dict = {
   'Best_City' : 'Krakow',
   'Best_University' : 'Jagiellonain',
   'Best_place' : 'market_square',
   'Best_food' : 'spicy',
   'Best_sport' : 'volleyball',
   'Year' : 2022
}
print(my_fist_dict)
```

One can access the values, using the key

```
print(my_first_dict['Best_place']
print(my_first_dict['Best_food']
```

- **After python 3.6**, **dictionaries are ordered**
- Dictionaries are **changeable** (change, add, remove) after dictionary is created.
- Unlike tuple, **dictionaries don't allow duplicates**

Dictionaries are very useful and flexible,
lets take another example

```
my_first_dict = {
    'Best_City' : ['Krakow', 'Warsaw', 'Lublin'],
    'Best_University' : 'Jagiellonain'
}
```

my_first_dict ['Best_City][2] ?

Adding, key-value pair in dictionary

my_first_dict['Best_habit'] = 'reading'

my_first_dict['year'] = 2022

Applying operation on values of given key

my_first_dict['year'] - 5 ?

Method name	Description
items( )	returns a list for each key value pair
keys( )	list out the dictionary keys
values( )	list out all the values in the dictionary
fromkeys( )	return the dictionary element for specified key:value
get( )	return the value of specified key
pop( )	removes the element with specified key
popitem( )	remove last inserted key-value pair
copy( )	returns a copy of the dictionary
update( )	updates the dictionary with the specified key-value
clear( )	removes all the elements from dictionary

Dictionaries are very useful and flexible,
lets take another example

```
my_first_dict = {
   'Best_City' : ['Krakow', 'Warsaw', 'Lublin'],
   'Best_University' : 'Jagiellonain'
}
```

my_first_dict ['Best_City][2] ?

Adding, key-value pair in dictionary

my_first_dict['Best_habit'] = 'reading'

my_first_dict['year'] = 2022

Applying operation on values of given key

my_first_dict['year'] - 5 ?

Method name	Description
items( )	**returns a list for each key value pair**
keys( )	**list out the dictionary keys**
values( )	**list out all the values in the dictionary**
fromkeys( )	return the dictionary element for specified key:value
get( )	**return the value of specified key**
pop( )	removes the element with specified key
popitem( )	**remove last inserted key-value pair**
copy( )	**returns a copy of the dictionary**
update( )	**updates the dictionary with the specified key-value**
clear( )	**removes all the elements from dictionary**

Dictionary : Hands-On

Keys in dictionary must be unique and of immutable types (strings, numbers, tuples)

One can define empty dictionary, unlike tuples - `fruit_inventory = {}`

Create list : `fruit_inventory = {'apple': 2, 'banana': 5, 'mango': 10}`

Using keys() and values() methods -

```
print("Keys:", fruit_inventory.keys())
print("Values:", fruit_inventory.values())
```

Using get methods to find value -

```
mango_quantity = fruit_inventory.get('mango')
```

Adding / updating key-value pairs -

```
fruit_inventory['orange'] = 7
fruit_inventory['apple'] = 4
print("Updated inventory:", fruit_inventory)
```

Removing key-value pairs -

```
fruit_inventory.pop('banana')
```

If keys are wrongly assigned, to rename it - `dict_new['key']=dict_new.pop('key_to_be_changed')`

Make dictionary from two lists, using - `New_dict = dict(zip(list1, list2))`

Try to merge the two dictionaries - `dict_new = {**dict1, **dict2}`

Access and change the nested dictionary element value - `Using dict_new[key][key]= value`

Dictionary : Hands-On

Keys in dictionary must be unique and of immutable types (strings, numbers, tuples)

One can define empty dictionary, unlike tuples - `fruit_inventory = {}`

Create list : `fruit_inventory = {'apple': 2, 'banana': 5, 'mango': 10}`

Using keys() and values() methods -

```
print("Keys:", fruit_inventory.keys())
print("Values:", fruit_inventory.values())
```

Using get methods to find value -

```
mango_quantity = fruit_inventory.get('mango')
```

Adding / updating key-value pairs -

```
fruit_inventory['orange'] = 7
fruit_inventory['apple'] = 4
print("Updated inventory:", fruit_inventory)
```

Removing key-value pairs -

```
fruit_inventory.pop('banana')
```

If keys are wrongly assigned, to rename it - *dict_new['key']=dict_new.pop('key_to_be_changed')*

Make dictionary from two lists, using - *New_dict = dict(zip(list1, list2))*

Try to merge the two dictionaries - *dict_new = {**dict1, **dict2}*

Access and change the nested dictionary element value - *Using dict_new[key][key]= value*

Dictionary : Hands-On

Keys in dictionary must be unique and of immutable types (strings, numbers, tuples)

One can define empty dictionary, unlike tuples - `fruit_inventory = {}`

Create list : `fruit_inventory = {'apple': 2, 'banana': 5, 'mango': 10}`

Using keys() and values() methods -

```
print("Keys:", fruit_inventory.keys())
print("Values:", fruit_inventory.values())
```

Using get methods to find value - `mango_quantity = fruit_inventory.get('mango')`

Adding / updating key-value pairs -

```
fruit_inventory['orange'] = 7
fruit_inventory['apple'] = 4
print("Updated inventory:", fruit_inventory)
```

Removing key-value pairs - `fruit_inventory.pop('banana')`

If keys are wrongly assigned, to rename it - *dict_new['key']=dict_new.pop('key_to_be_changed')*

Make dictionary from two lists, using - *New_dict = dict(zip(list1, list2))*

Try to merge the two dictionaries - *dict_new = {**dict1, **dict2}*

Access and change the nested dictionary element value - *Using dict_new[key][key]= value*

Dictionary : Hands-On

Keys in dictionary must be unique and of immutable types (strings, numbers, tuples)

One can define empty dictionary, unlike tuples - `fruit_inventory = {}`

Create list : `fruit_inventory = {'apple': 2, 'banana': 5, 'mango': 10}`

Using keys() and values() methods -

```
print("Keys:", fruit_inventory.keys())
print("Values:", fruit_inventory.values())
```

Using get methods to find value -

```
mango_quantity = fruit_inventory.get('mango')
```

Adding / updating key-value pairs -

```
fruit_inventory['orange'] = 7
fruit_inventory['apple'] = 4
print("Updated inventory:", fruit_inventory)
```

Removing key-value pairs -

```
fruit_inventory.pop('banana')
```

If keys are wrongly assigned, to rename it - `dict_new['key']=dict_new.pop('key_to_be_changed')`

Make dictionary from two lists, using - `New_dict = dict(zip(list1, list2))`

Try to merge the two dictionaries - `dict_new = {**dict1, **dict2}`

Access and change the nested dictionary element value - *Using dict_new[key][key]= value*

Dictionary : Hands-On

Keys in dictionary must be unique and of immutable types (strings, numbers, tuples)

One can define empty dictionary, unlike tuples - `fruit_inventory = {}`

Create list : `fruit_inventory = {'apple': 2, 'banana': 5, 'mango': 10}`

Using keys() and values() methods -

```
print("Keys:", fruit_inventory.keys())
print("Values:", fruit_inventory.values())
```

Using get methods to find value -

```
mango_quantity = fruit_inventory.get('mango')
```

Adding / updating key-value pairs -

```
fruit_inventory['orange'] = 7
fruit_inventory['apple'] = 4
print("Updated inventory:", fruit_inventory)
```

Removing key-value pairs - `fruit_inventory.pop('banana')`

If keys are wrongly assigned, to rename it - *dict_new['key']=dict_new.pop('key_to_be_changed')*

Make dictionary from two lists, using - *New_dict = dict(zip(list1, list2))*

Try to merge the two dictionaries - *dict_new = {**dict1, **dict2}*

Access and change the nested dictionary element value - *Using dict_new[key][key]= value*

Dictionary : Hands-On

Keys in dictionary must be unique and of immutable types (strings, numbers, tuples)

One can define empty dictionary, unlike tuples - `fruit_inventory = {}`

Create list : `fruit_inventory = {'apple': 2, 'banana': 5, 'mango': 10}`

Using keys() and values() methods -

```
print("Keys:", fruit_inventory.keys())
print("Values:", fruit_inventory.values())
```

Using get methods to find value -

```
mango_quantity = fruit_inventory.get('mango')
```

Adding / updating key-value pairs -

```
fruit_inventory['orange'] = 7
fruit_inventory['apple'] = 4
print("Updated inventory:", fruit_inventory)
```

Removing key-value pairs -

```
fruit_inventory.pop('banana')
```

If **keys are wrongly assigned**, to rename it - *dict_new[‘key’]=dict_new.**pop**(‘key_to_be_changed’)*

Make dictionary from two lists, using - *New_dict = dict(zip(list1, list2))*

Try to merge the two dictionaries - *dict_new = {**dict1, **dict2}*

Access and change the nested dictionary element value - *Using dict_new[key][key]= value*

Dictionary : Hands-On

Keys in dictionary must be unique and of immutable types (strings, numbers, tuples)

One can define empty dictionary, unlike tuples - `fruit_inventory = {}`

Create list : `fruit_inventory = {'apple': 2, 'banana': 5, 'mango': 10}`

Using keys() and values() methods -

```
print("Keys:", fruit_inventory.keys())
print("Values:", fruit_inventory.values())
```

Using get methods to find value -

```
mango_quantity = fruit_inventory.get('mango')
```

Adding / updating key-value pairs -

```
fruit_inventory['orange'] = 7
fruit_inventory['apple'] = 4
print("Updated inventory:", fruit_inventory)
```

Removing key-value pairs -

```
fruit_inventory.pop('banana')
```

If **keys are wrongly assigned**, to rename it - *dict_new['key']=dict_new.**pop**('key_to_be_changed')*

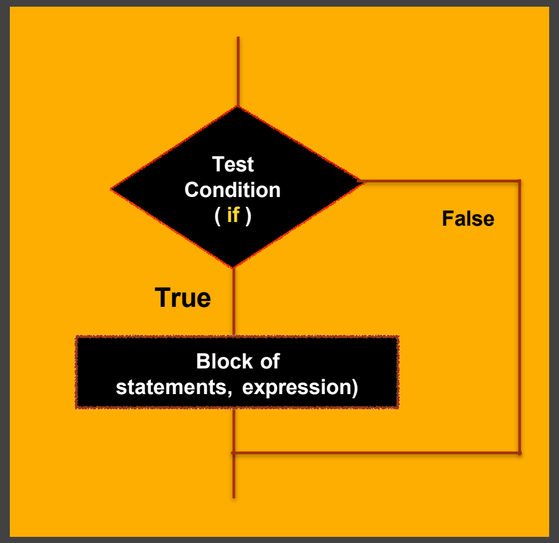
Make dictionary from two lists, using - *New_dict = **dict**(**zip**(list1, list2))*

Try to merge the **two dictionaries** - *dict_new = {**dict1, **dict2}*

Access and change the nested dictionary element value - *Using dict_new[key][key]= value*

If, if-else, elif statements (decision making building blocks in Python)

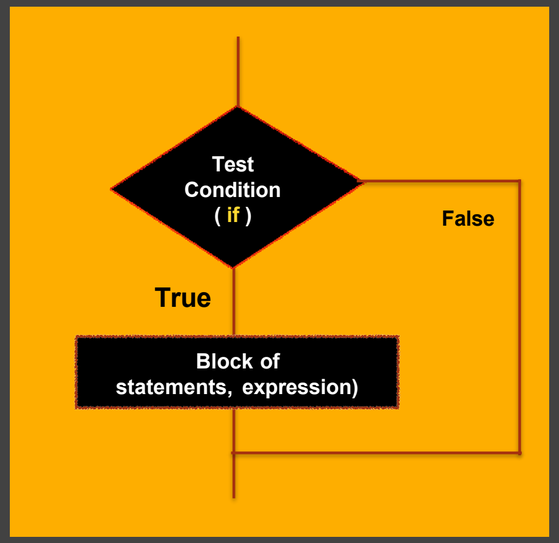
If Statement




```
x = 10
If x > 10:
    print(“ Number is Greater”)
```

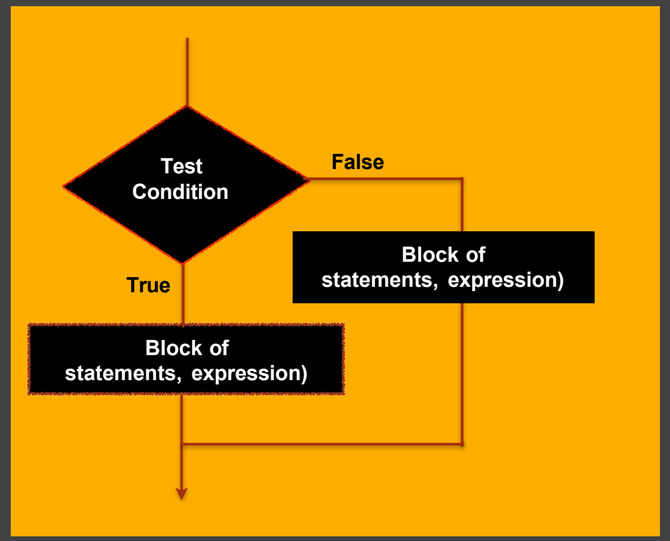
If, if-else, elif statements (decision making building blocks in Python)

If Statement




```
x = 10
If x > 10:
    print(“ Number is Greater”)
```

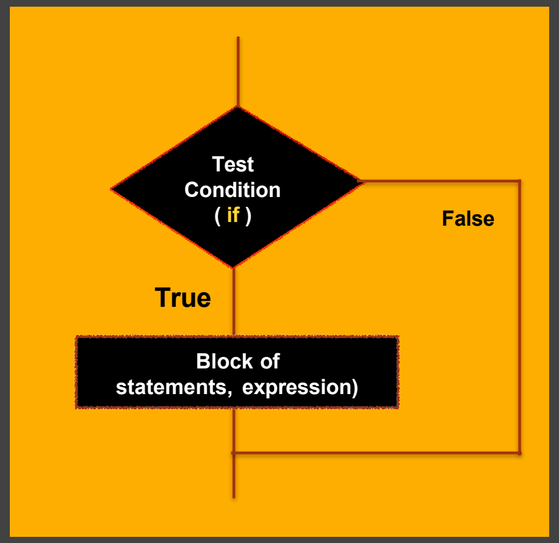
If-else Statement




```
If x > 10:
    print(“Number is Greater”)
else:
    print(“Number is smaller”)
```

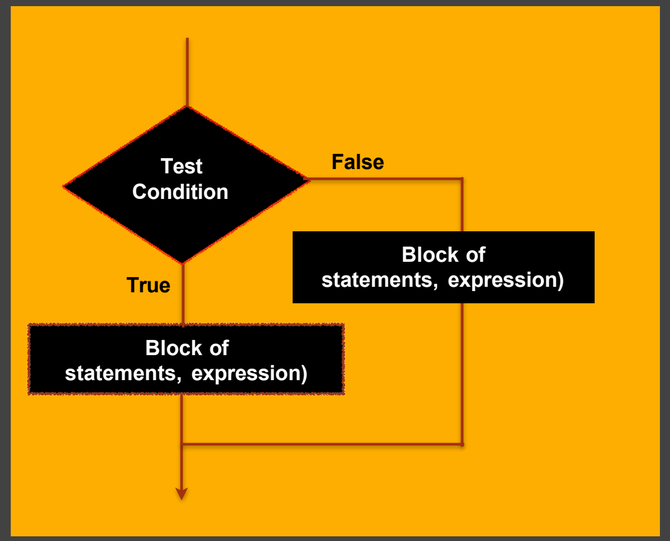
If, if-else, elif statements (decision making building blocks in Python)

If Statement



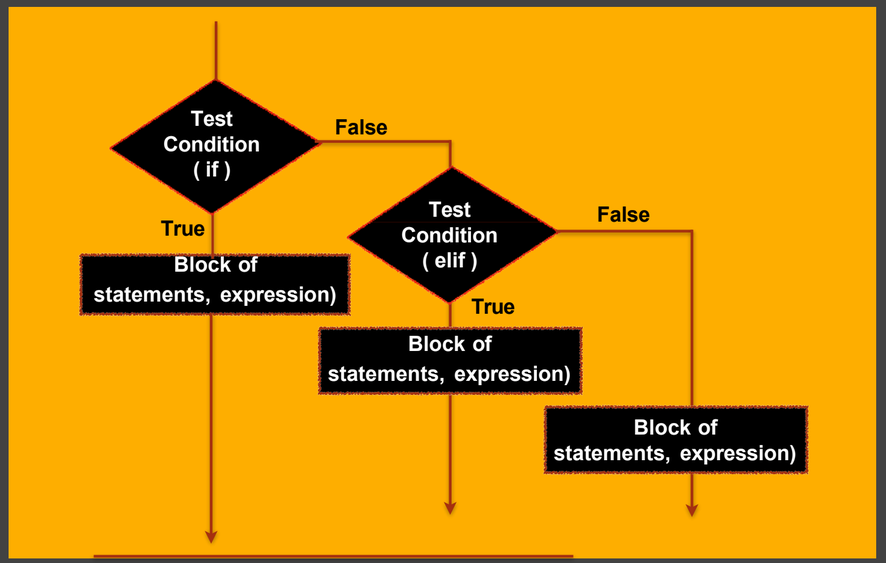

```
x = 10
If x > 10:
    print(“ Number is Greater”)
```

If-else Statement




```
If x > 10:
    print(“Number is Greater”)
else:
    print(“Number is smaller”)
```

If-elif Statement




```
If x > 10:
    print(“Number is Greater”)
elif x == 10:
    print(“Number is smaller”)
else:
    print(“Number is smaller”)
```

Looping in Python

After decision making,

LOOPs are another key feature of any fundamental language, that allows to execute the same action several times.

Print integer numbers between 1 to 10 ?

```
print(1)
1
print(2)
2
print(3)
3
.
.
.print(10)
10
```

Looping in Python

After decision making,

LOOPs are another key feature of any fundamental language, that allows to execute the same action several times.

Print integer numbers between 1 to 10 ?

```
print(1)
1
print(2)
2
print(3)
3
.
.
.print(10)
10
```

Using LOOP :

```
for x in range(1,11):
    print(x)
```

Print only odd / Even numbers ?

```
for letter in "coffee":
    print(letter*10)
```

What it does ?

-Assignments-

Assignments