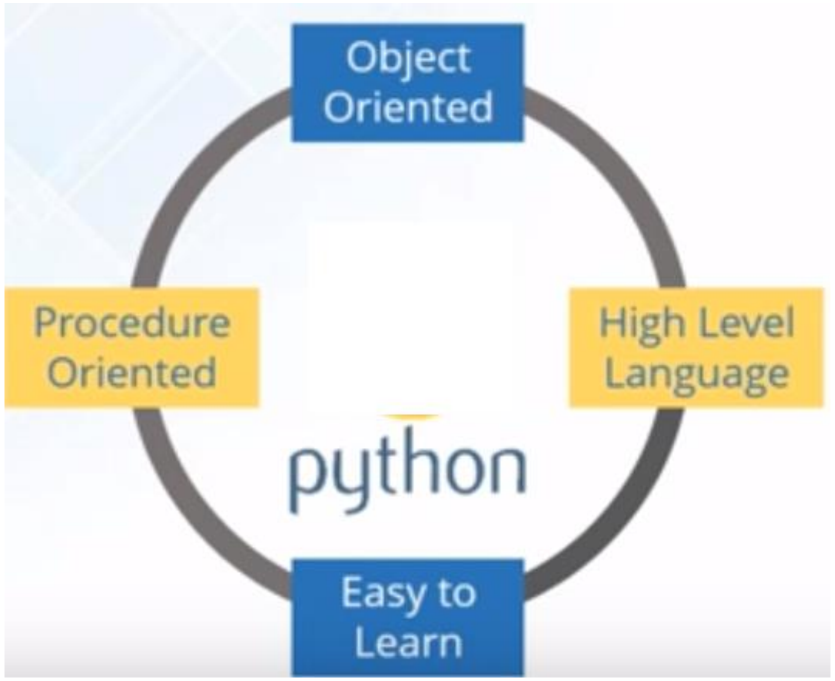
Python Programming

-Raghul

01-Oct-2018

Python Introduction:

- Python was created by Guido Rossum 1989 and is very easy to learn
- Python is an Interpreted , Object Oriented , high-level programming language with dynamic semantics



Who uses Python:

- Youtube
- Google
- Dopbox
- BitTorrent
- NASA
- NETFLIX

Python Features

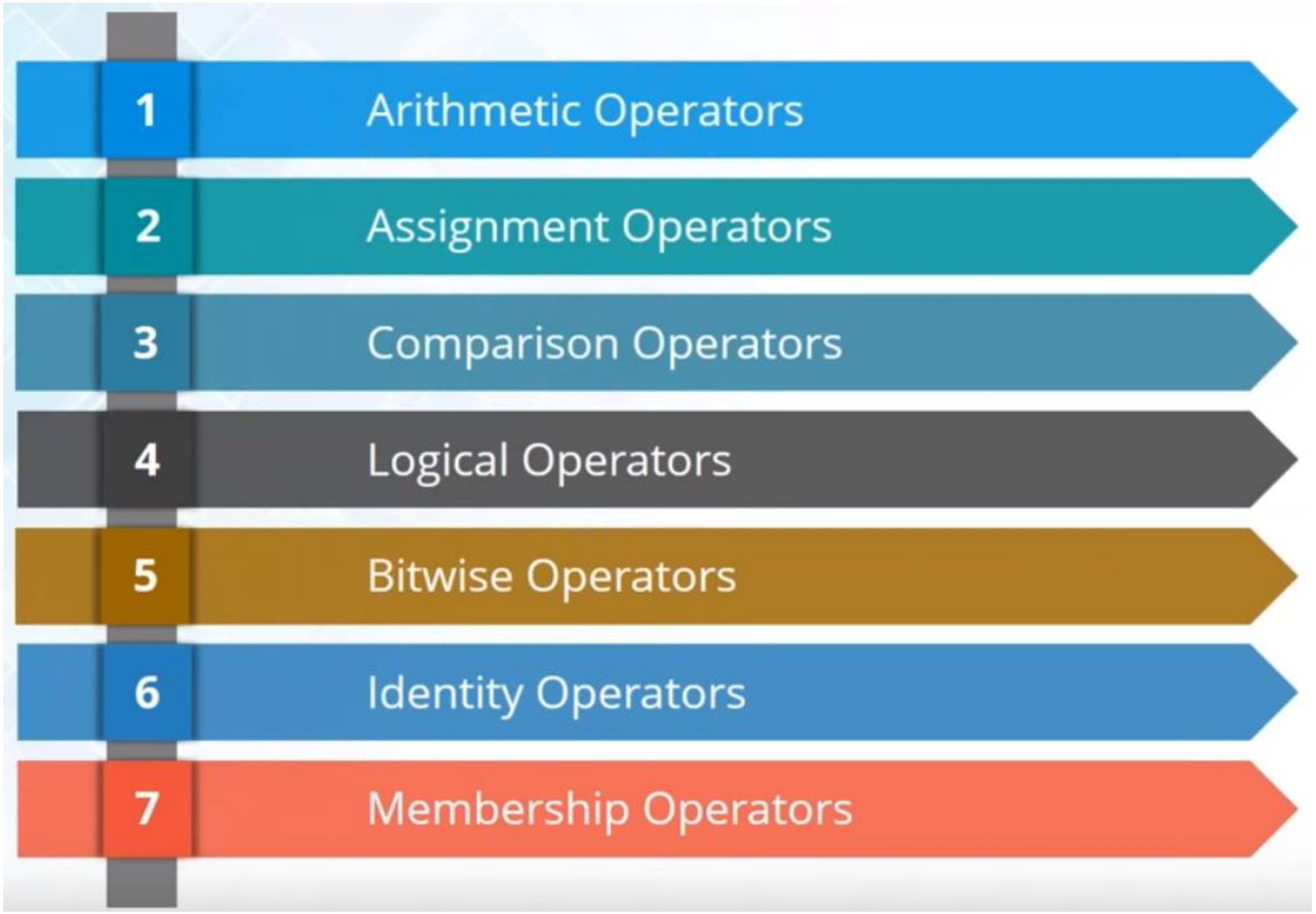
- Simple and Easy to learn
- Free and Open Source
- High Level Language
- Portable
- Supports different Programming paradigm
- Extensible

Python Installation

http://www.python.org

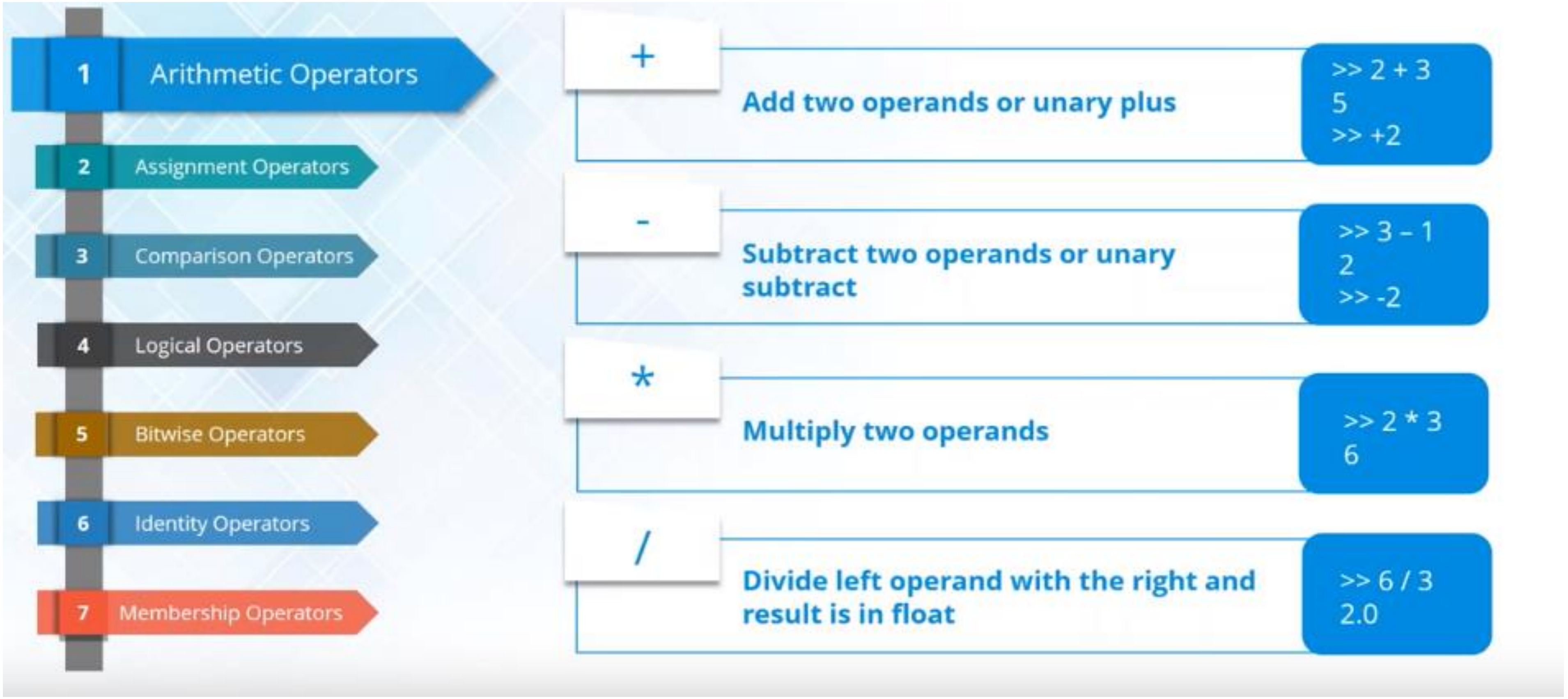
https://www.anaconda.com/download/#windows

Operators in Python:


1 Arithmetic Operators
2 Assignment Operators
3 Comparison Operators
4 Logical Operators
5 Bitwise Operators
6 Identity Operators
7 Membership Operators

Arithmetic Operators

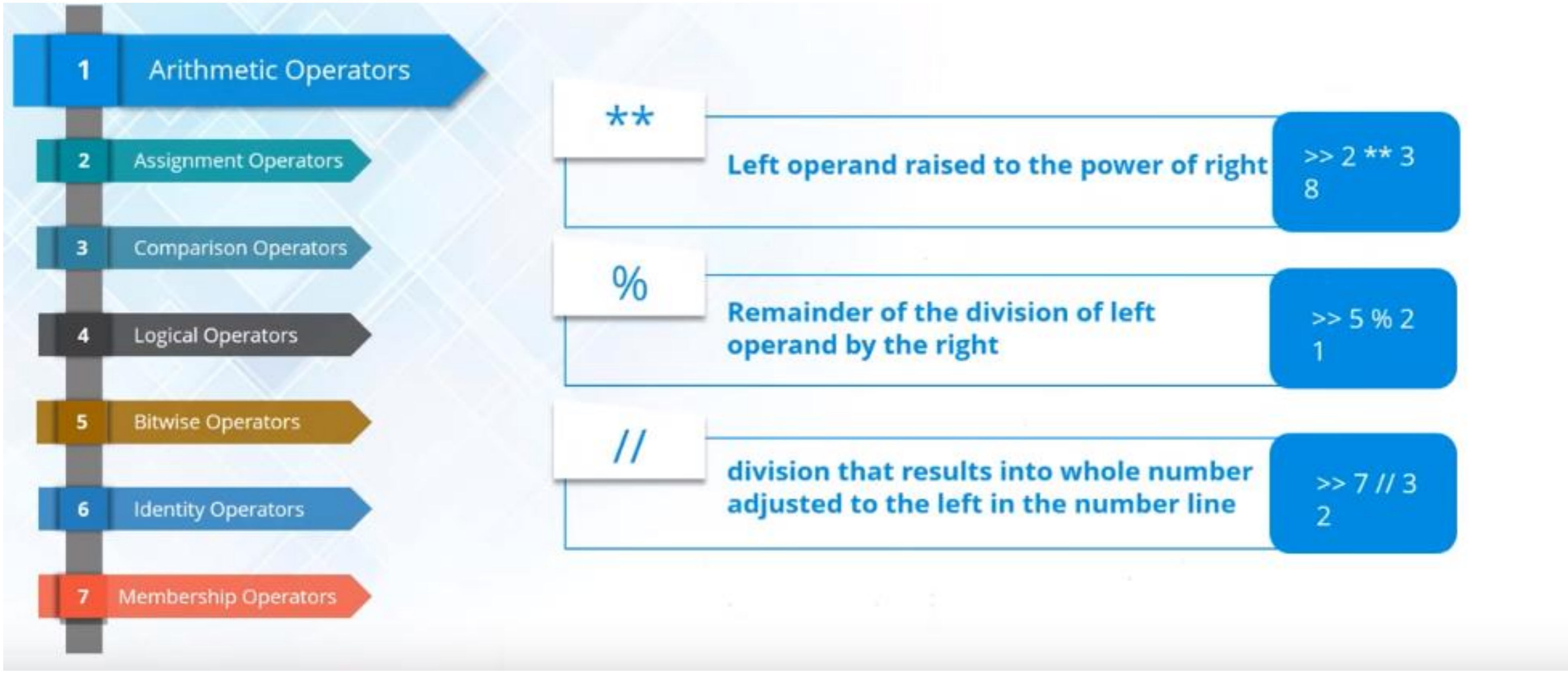
▪


1 Arithmetic Operators
2 Assignment Operators
3 Comparison Operators
4 Logical Operators
5 Bitwise Operators
6 Identity Operators
7 Membership Operators

Operator	Description	Example
+	Add two operands or unary plus	>> 2 + 3 5 >> +2
-	Subtract two operands or unary subtract	>> 3 – 1 2 >> -2
*	Multiply two operands	>> 2 * 3 6
/	Divide left operand with the right and result is in float	>> 6 / 3 2.0

Arithmetic Operators Cont…

▪



1 Arithmetic Operators

2 Assignment Operators

3 Comparison Operators

4 Logical Operators

5 Bitwise Operators

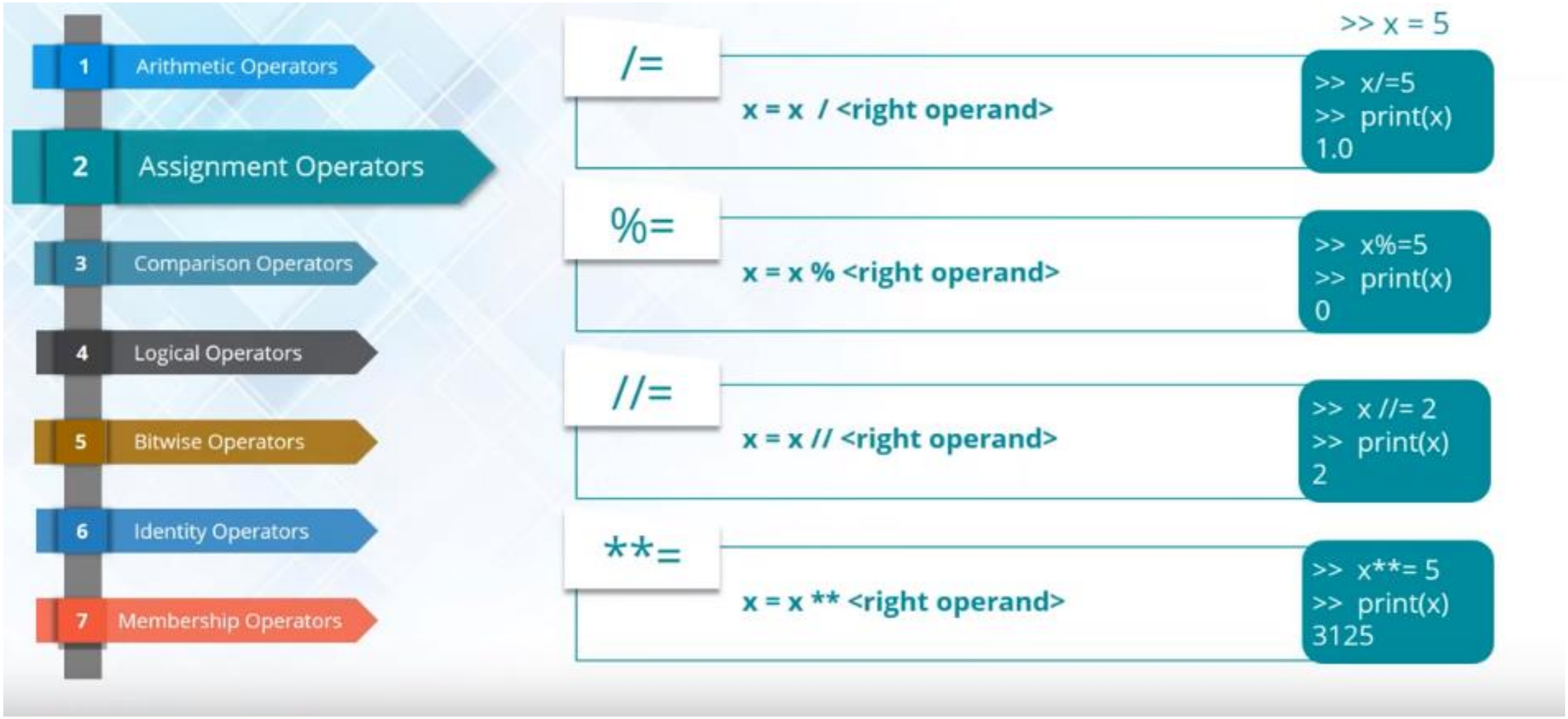
6 Identity Operators

7 Membership Operators

**	Left operand raised to the power of right	>> 2 ** 3 8
%	Remainder of the division of left operand by the right	>> 5 % 2 1
//	division that results into whole number adjusted to the left in the number line	>> 7 // 3 2

Assignment Operators

▪



```
>> x = 5
```

/=

x = x / <right operand>

```
>>  x/=5
>>  print(x)
1.0
```

%=

x = x % <right operand>

```
>>  x%=5
>>  print(x)
0
```

//=

x = x // <right operand>

```
>>  x //= 2
>>  print(x)
2
```

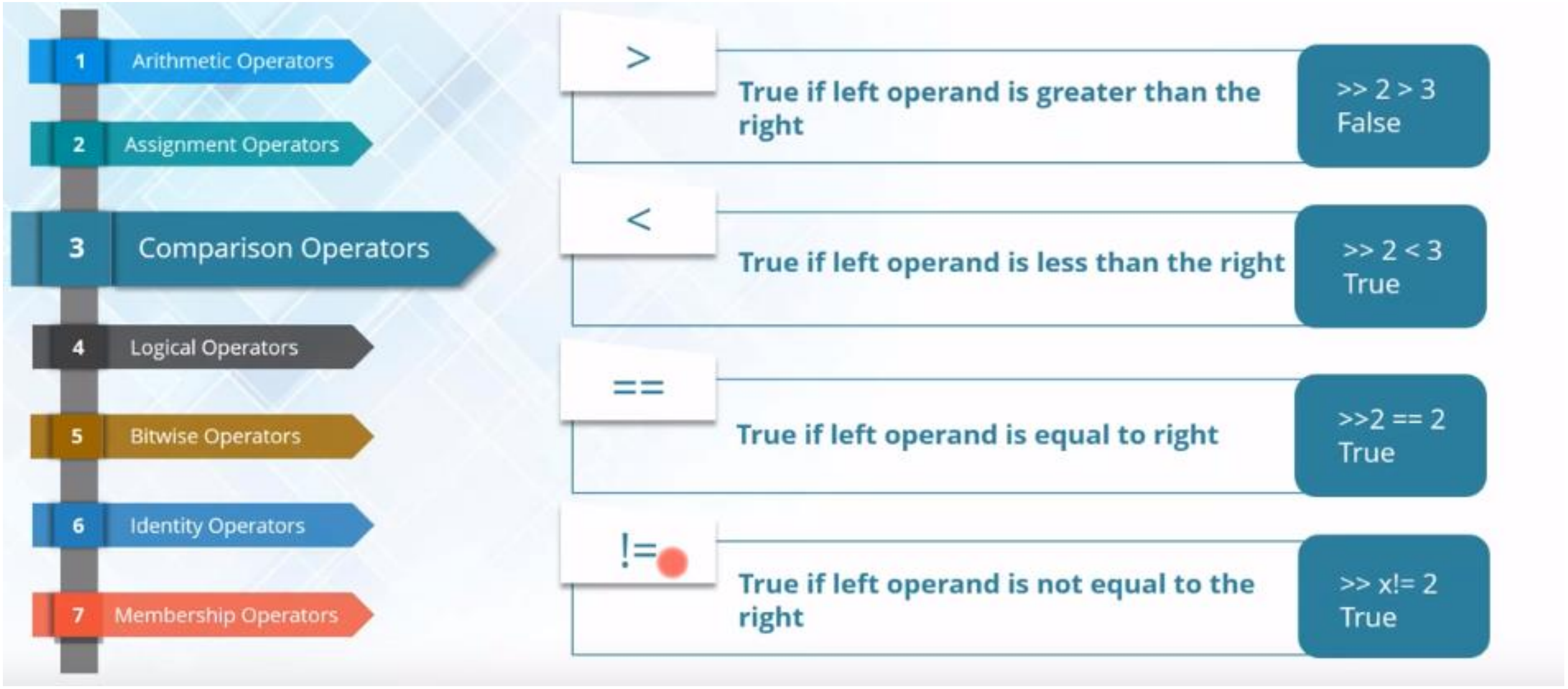
**=

x = x ** <right operand>

```
>>  x**= 5
>>  print(x)
3125
```

Comparison Operators

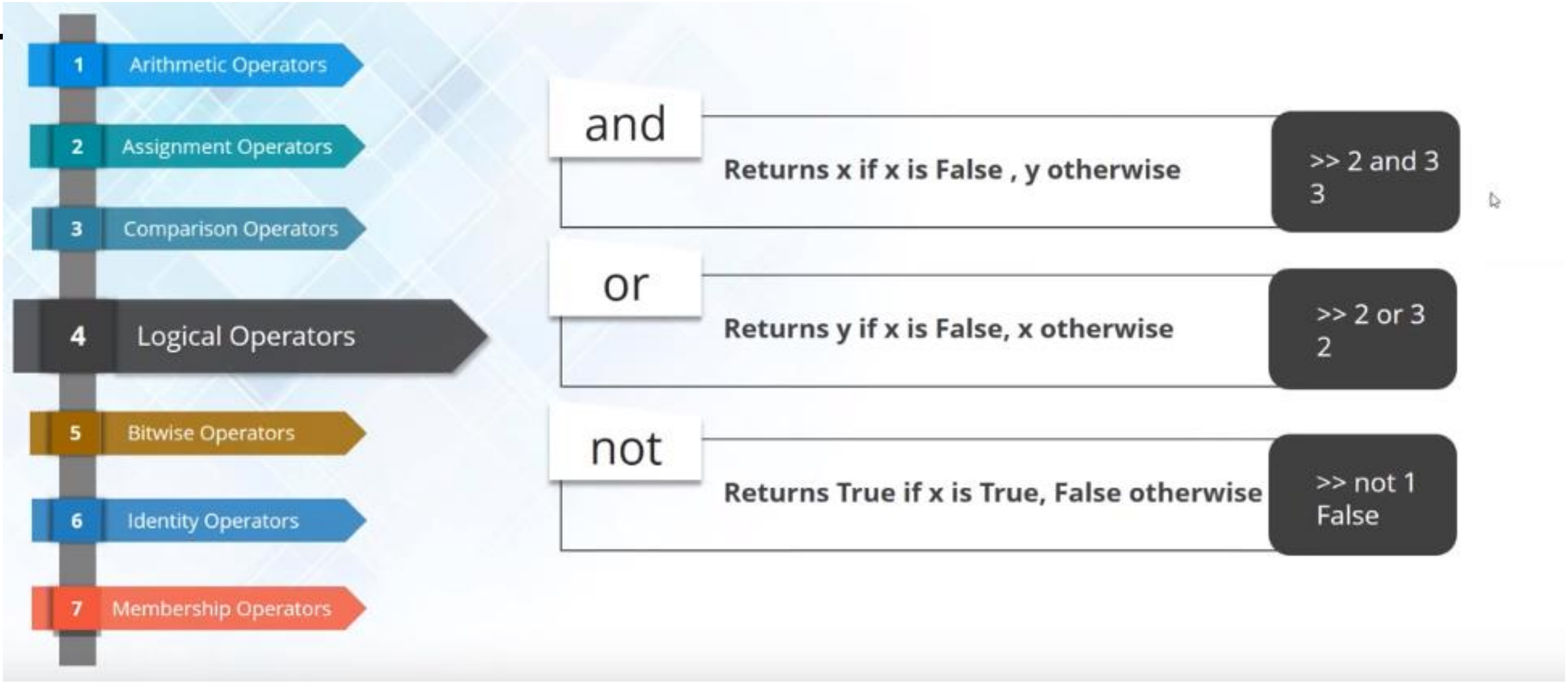
▪



1 Arithmetic Operators
2 Assignment Operators
3 Comparison Operators
4 Logical Operators
5 Bitwise Operators
6 Identity Operators
7 Membership Operators

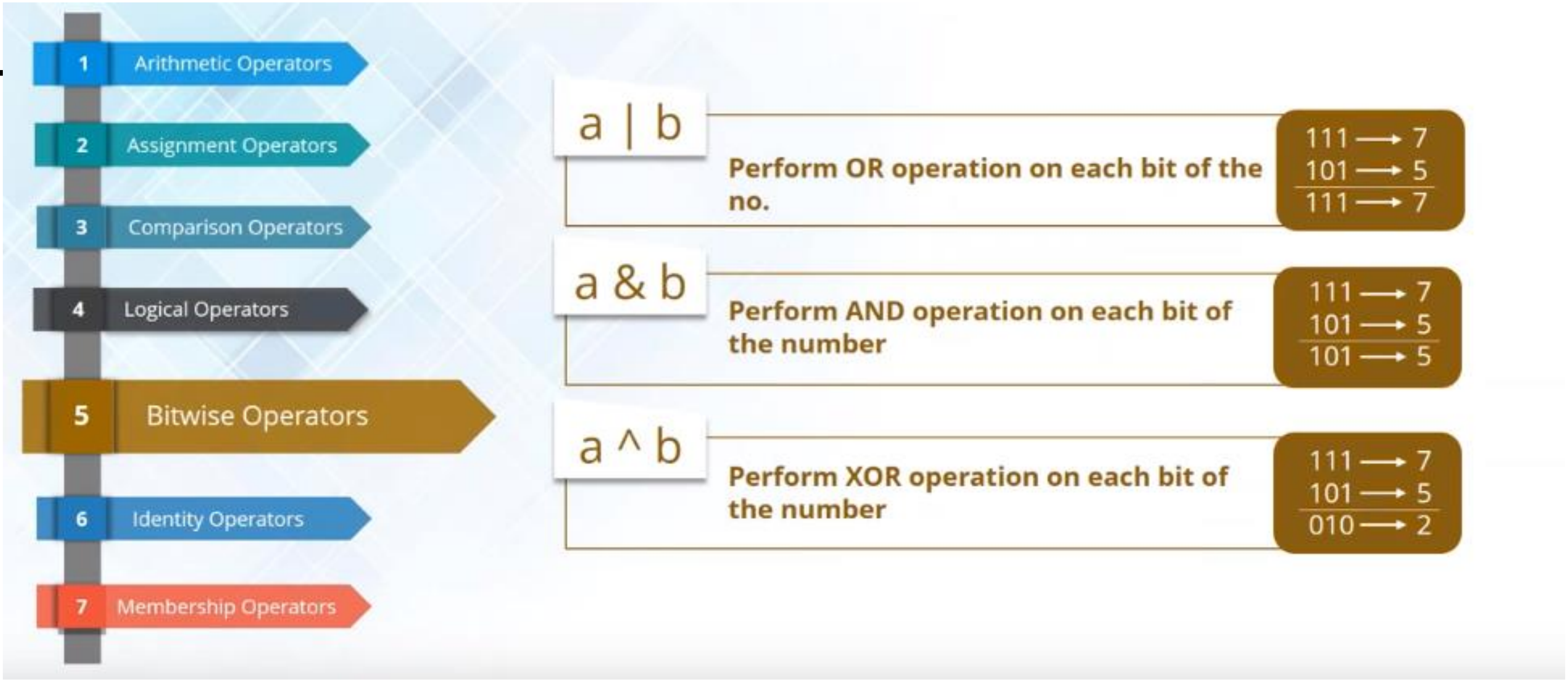
>	True if left operand is greater than the right	>> 2 > 3 False
<	True if left operand is less than the right	>> 2 < 3 True
==	True if left operand is equal to right	>>2 == 2 True
!=	True if left operand is not equal to the right	>> x!= 2 True

Logical Operators


1 Arithmetic Operators
2 Assignment Operators
3 Comparison Operators
4 Logical Operators
5 Bitwise Operators
6 Identity Operators
7 Membership Operators

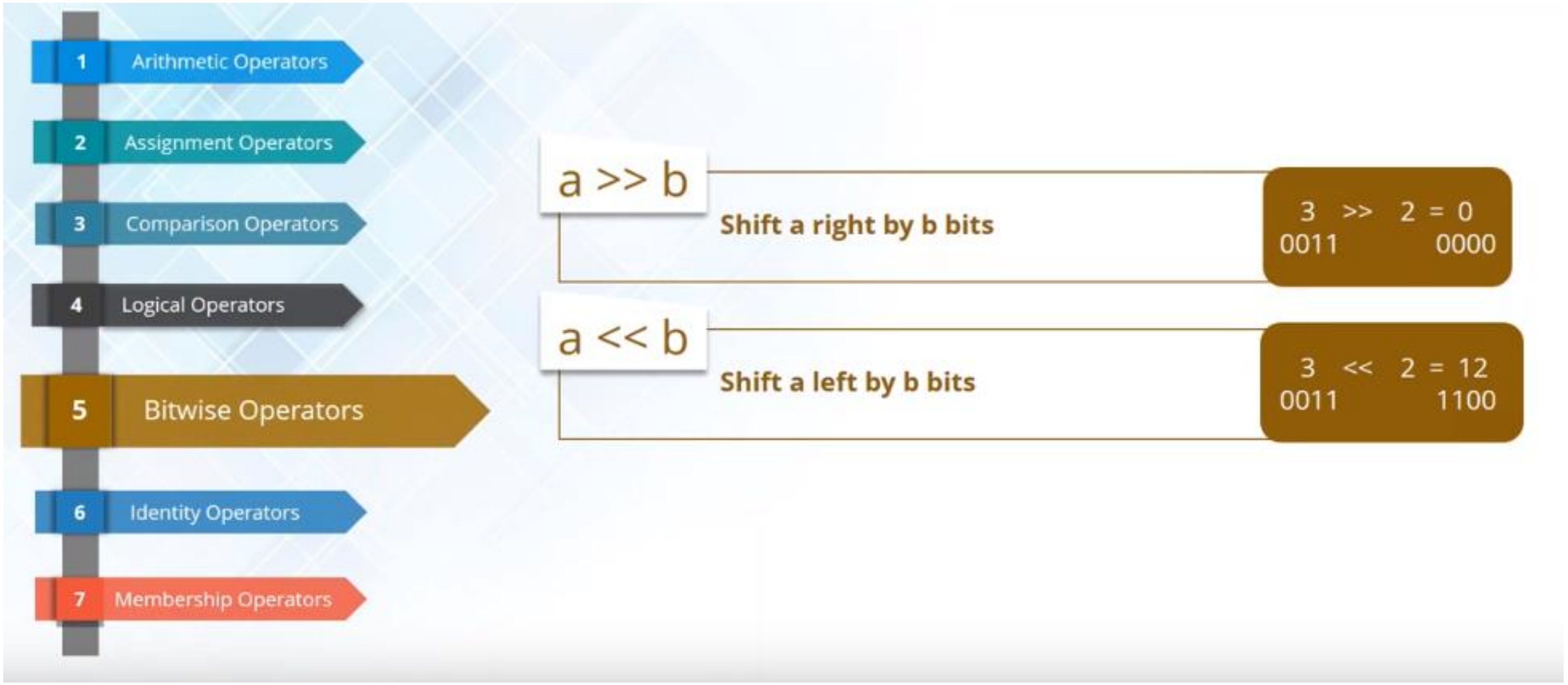
and	Returns x if x is False , y otherwise	>> 2 and 3 3
or	Returns y if x is False, x otherwise	>> 2 or 3 2
not	Returns True if x is True, False otherwise	>> not 1 False

Bitwise Operators

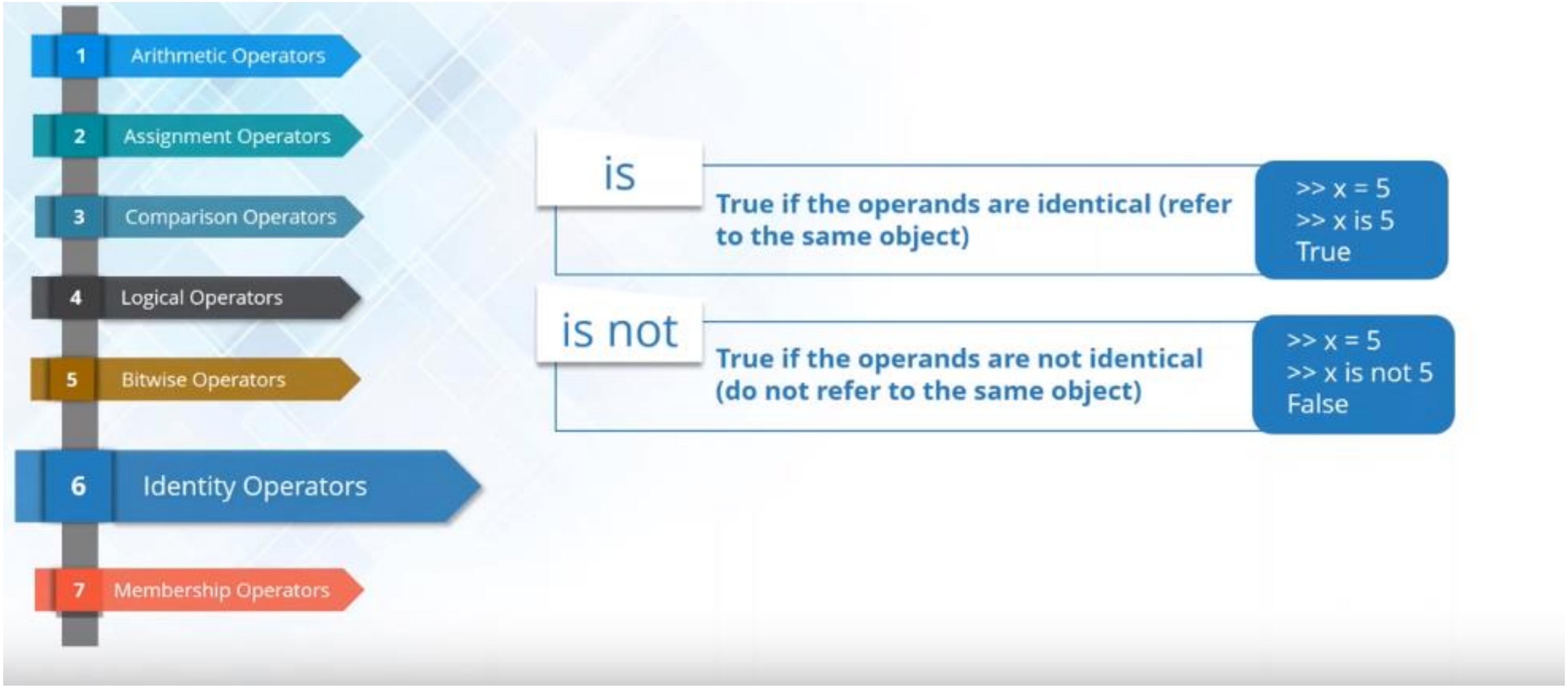


Bitwise Operators Cont…

▪



Identity Operators:



is — True if the operands are identical (refer to the same object)

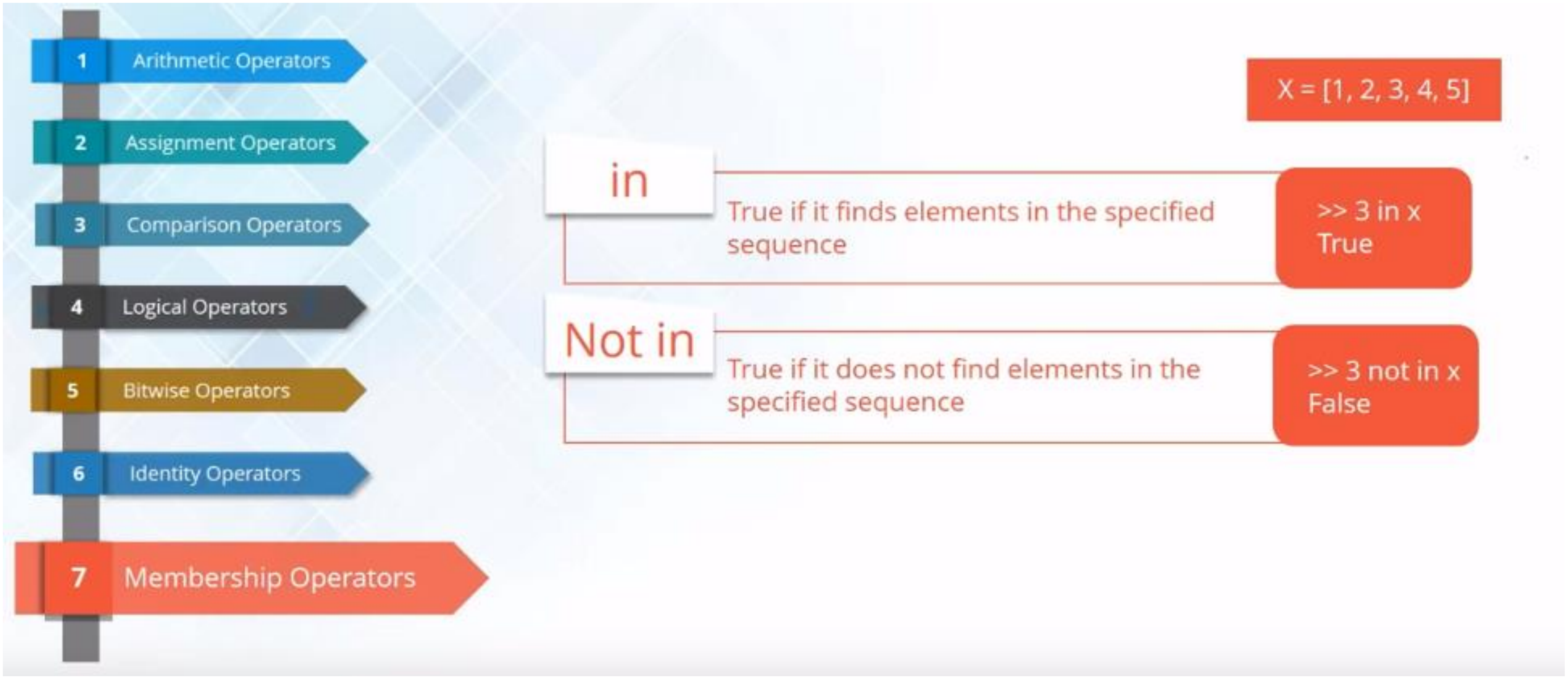
```
>> x = 5
>> x is 5
True
```

is not — True if the operands are not identical (do not refer to the same object)

```
>> x = 5
>> x is not 5
False
```

Membership Operators:

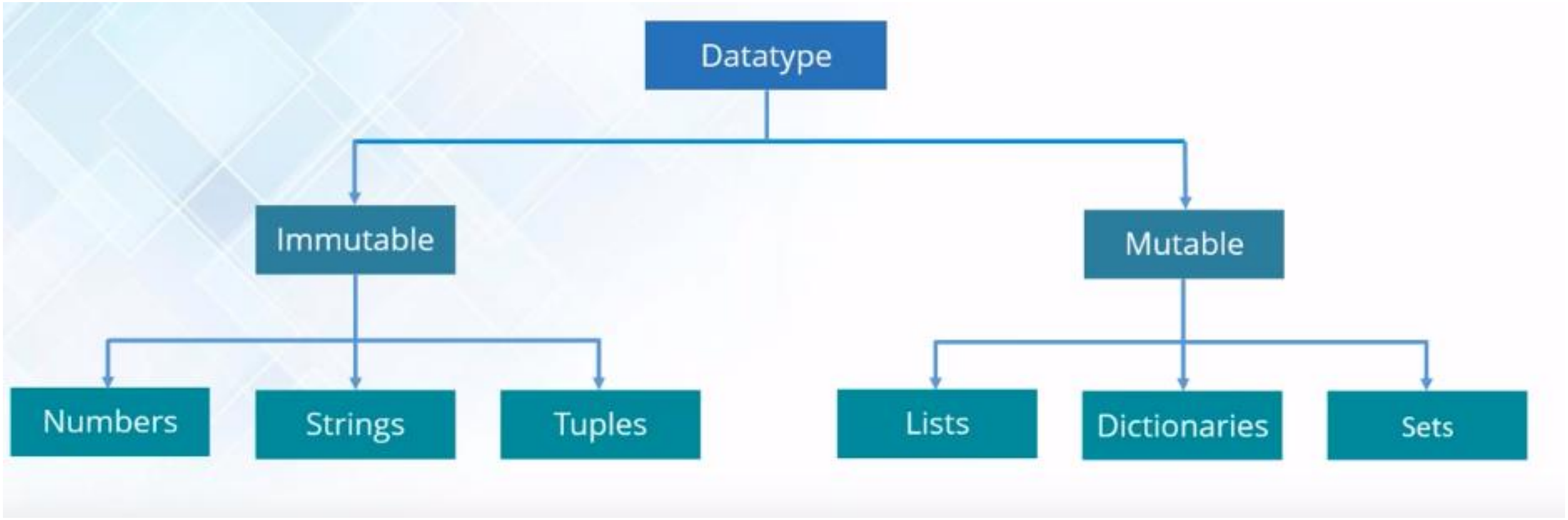
▪



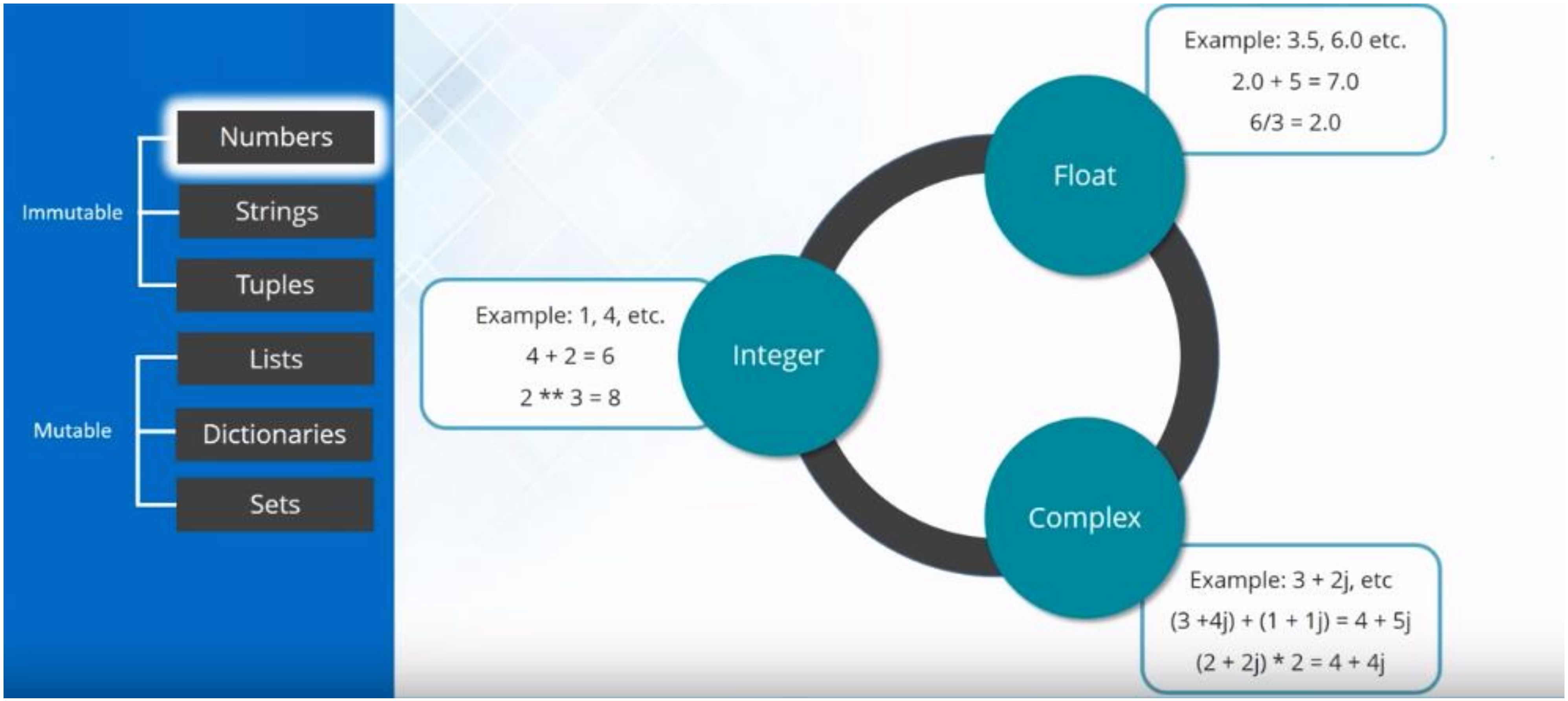
Datatype:

- . Python is loosely typed language. Therefore no need of define the data type of variables

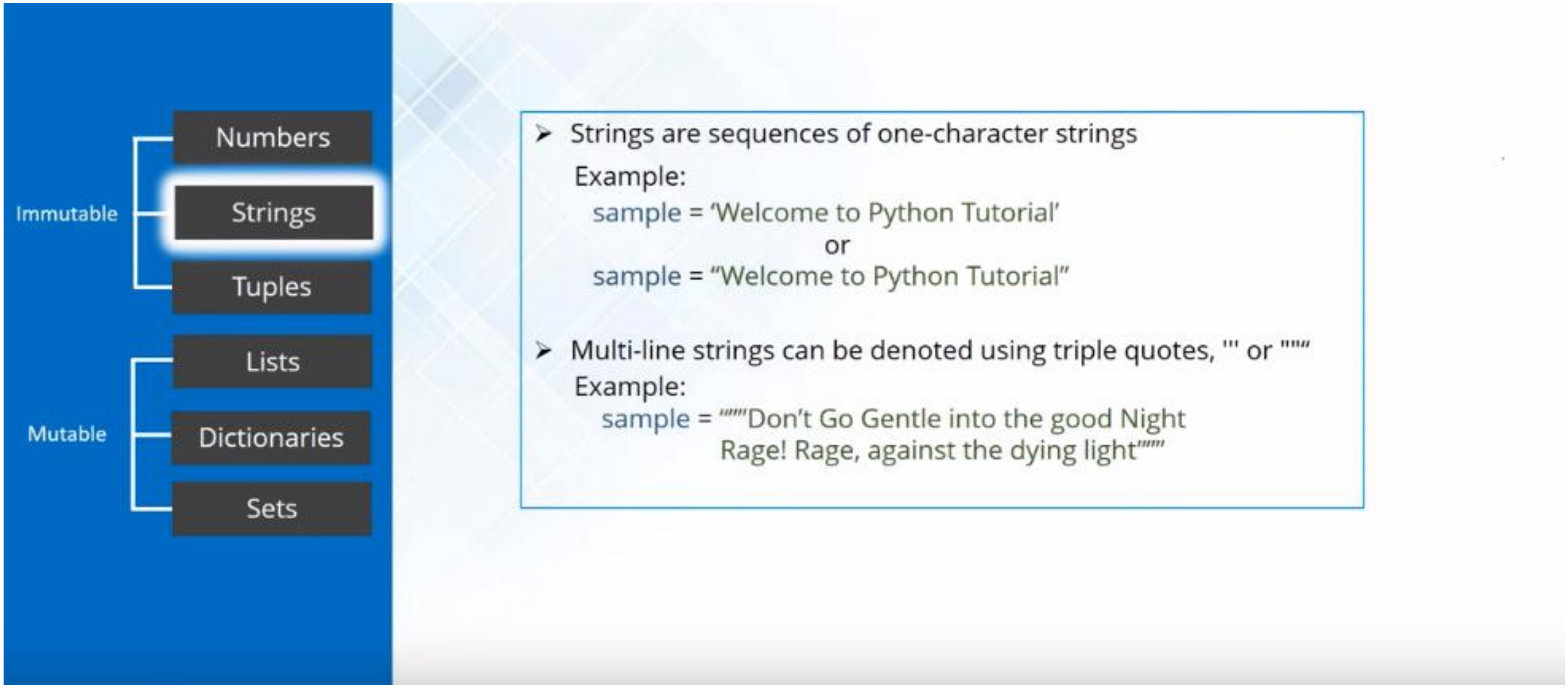
- No need to declare variables before using them



Datatype Cont…



Datatype Cont…



- Strings are sequences of one-character strings

  Example:

  sample = 'Welcome to Python Tutorial'

  or

  sample = "Welcome to Python Tutorial"

- Multi-line strings can be denoted using triple quotes, ''' or """

  Example:

  sample = """Don't Go Gentle into the good Night
  Rage! Rage, against the dying light"""

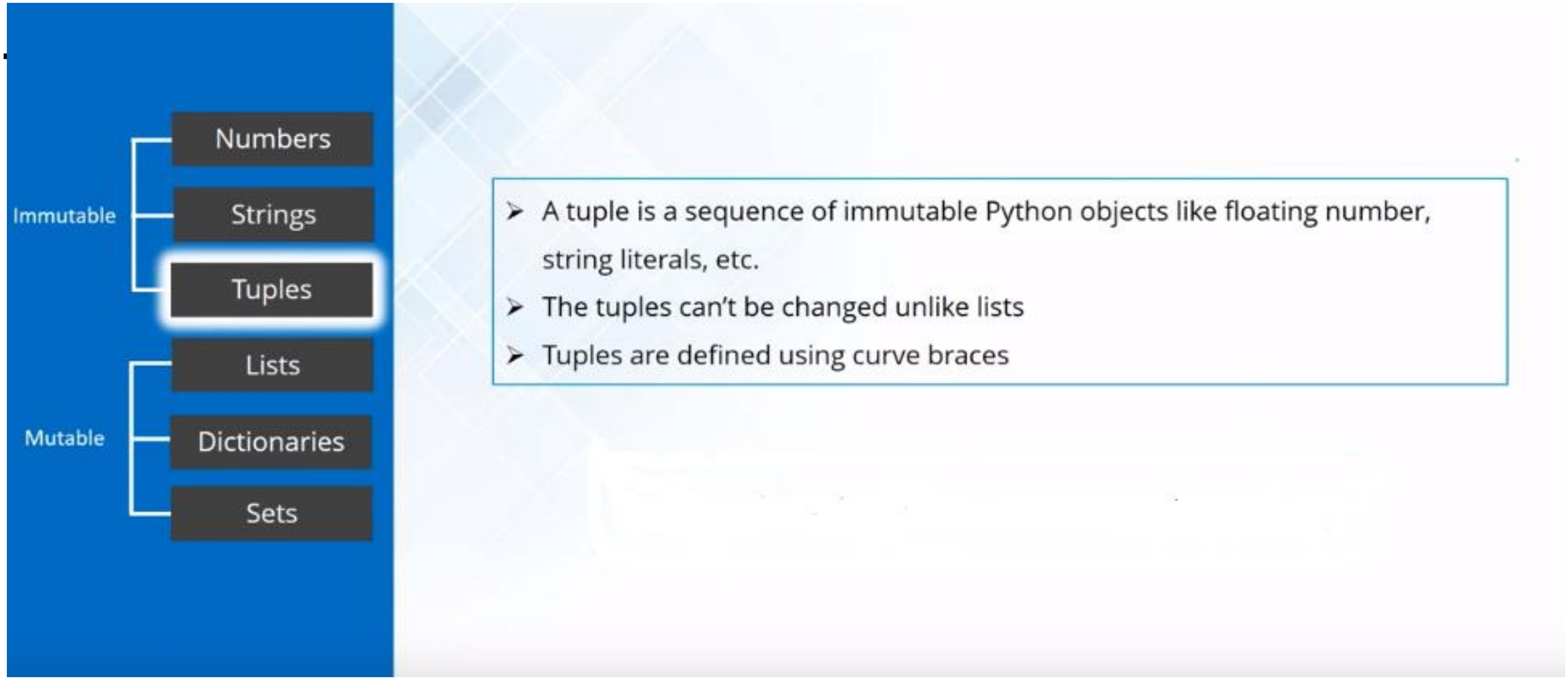
String Sequence Operations:

- Concatenation (+)
- Repetition(*)
- Slicing [start:end]
- Indexing [Position]

String Type Specific Method:

- find()
- replace()
- split()
- count()
- upper()
- max()
- min( )
- Isalpha()

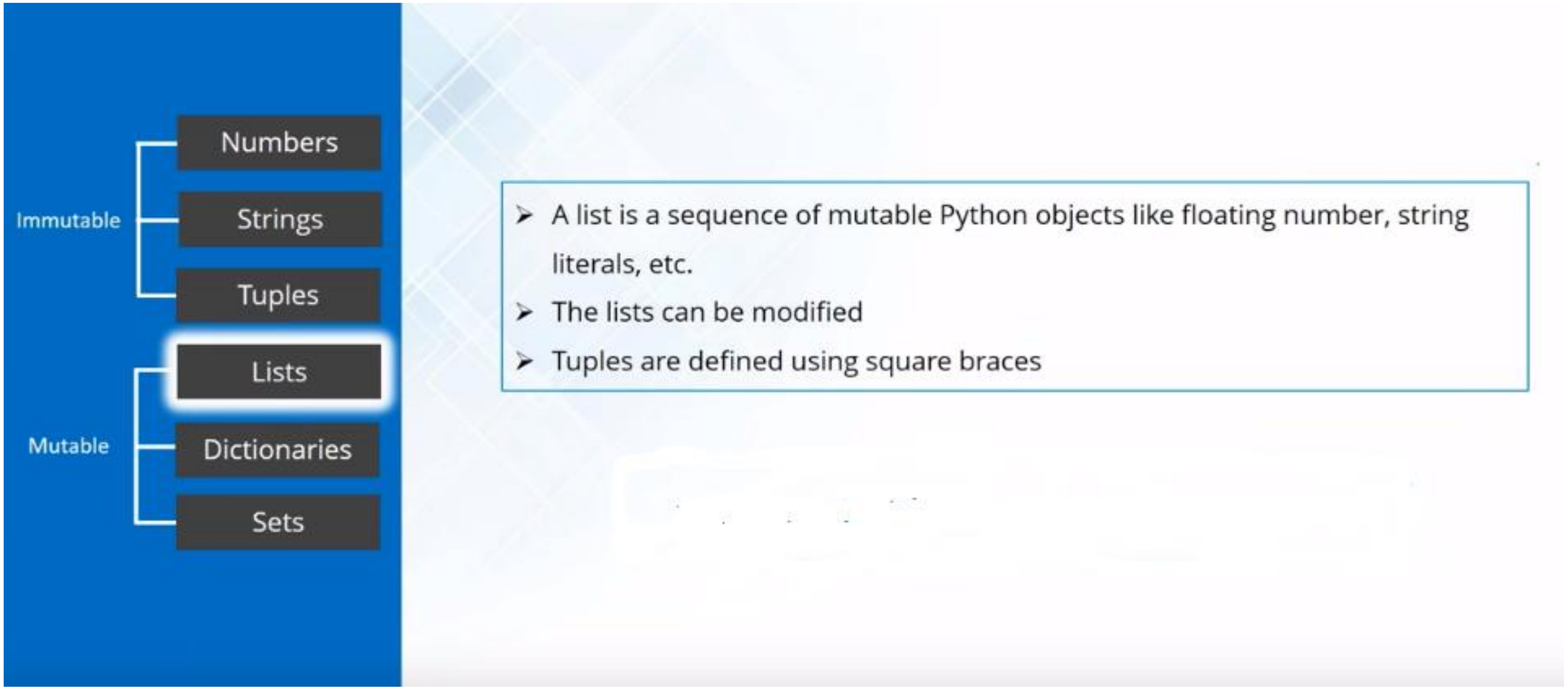
Datatype Cont…



- A tuple is a sequence of immutable Python objects like floating number, string literals, etc.
- The tuples can't be changed unlike lists
- Tuples are defined using curve braces

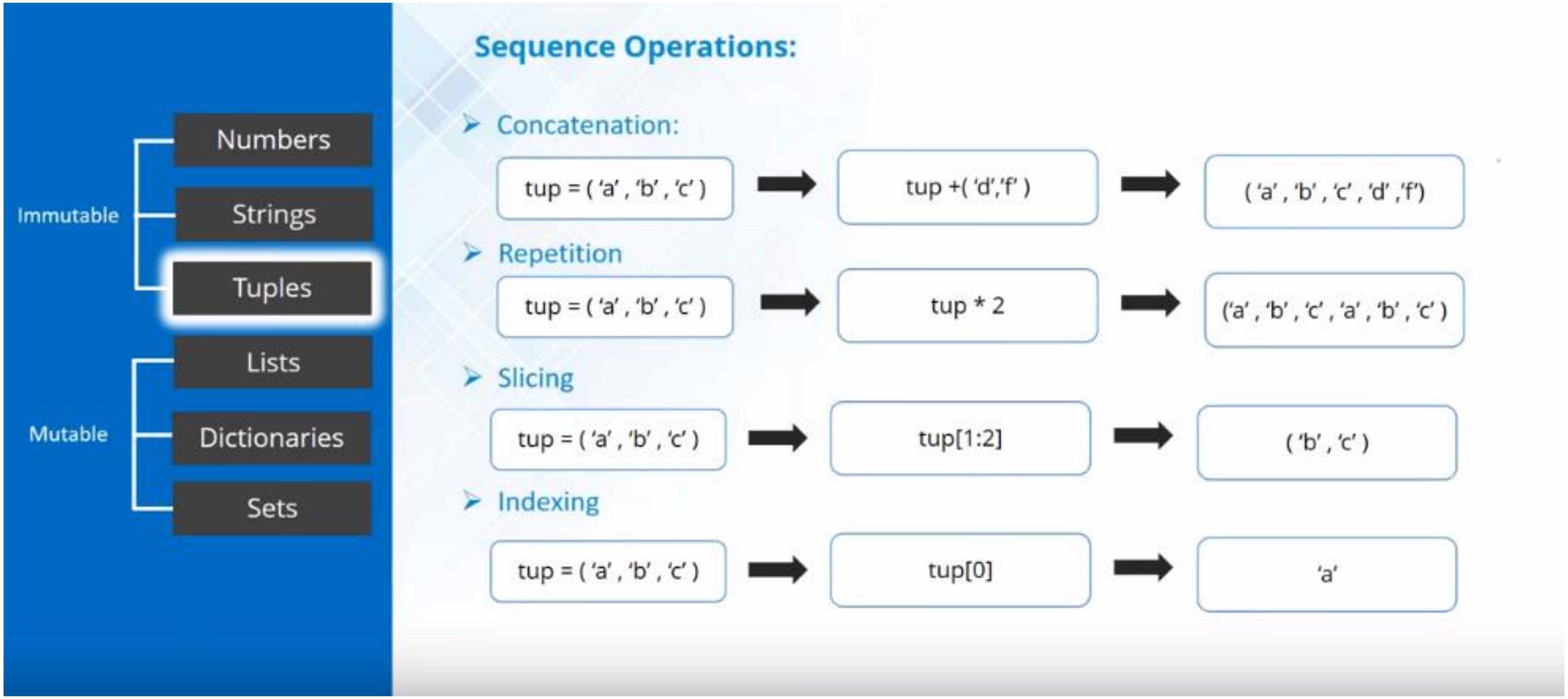
Datatype Cont…

▪



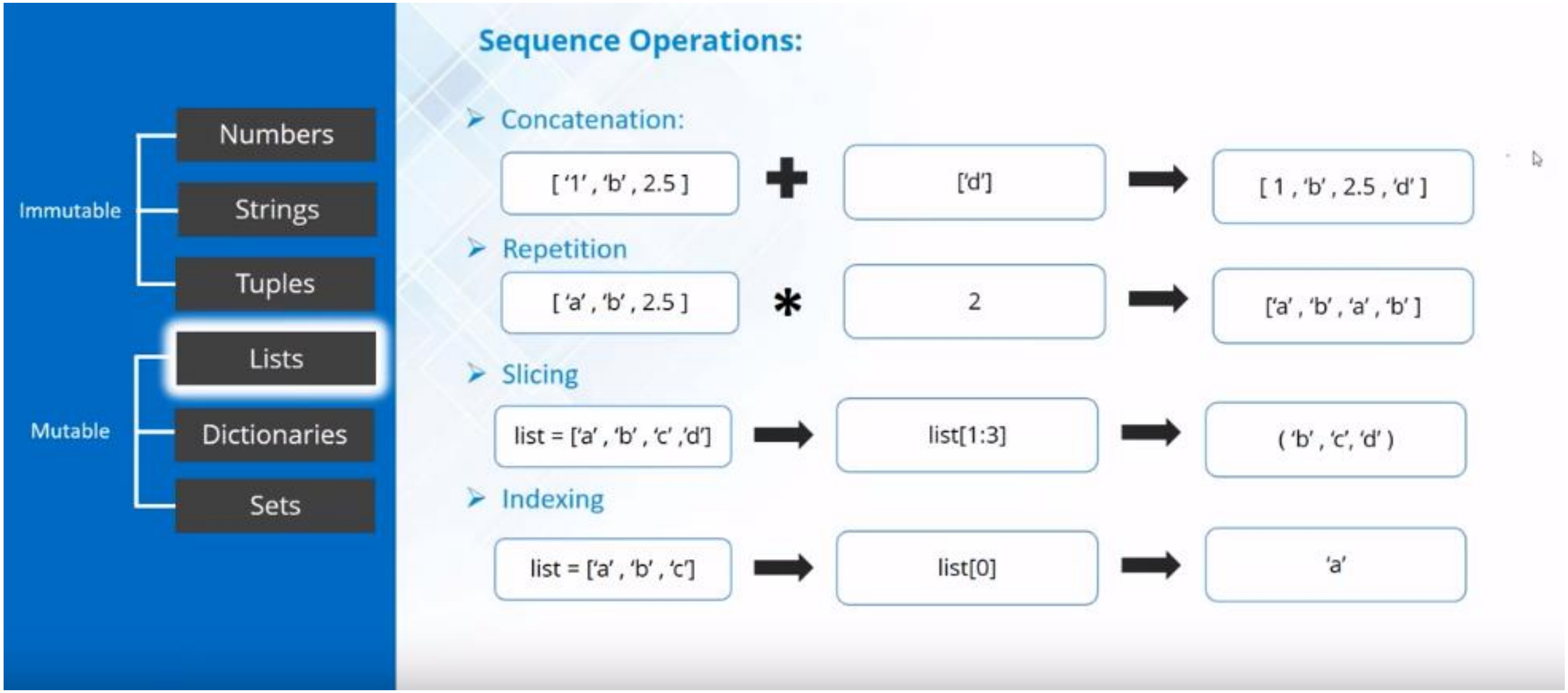
- A list is a sequence of mutable Python objects like floating number, string literals, etc.
- The lists can be modified
- Tuples are defined using square braces

Tuple Operations:



List Operations:

▪



Sequence Operations:

➢ Concatenation:

['1' , 'b' , 2.5] ✚ ['d'] ➡ [1 , 'b' , 2.5 , 'd']

➢ Repetition

['a' , 'b' , 2.5] ✱ 2 ➡ ['a' , 'b' , 'a' , 'b']

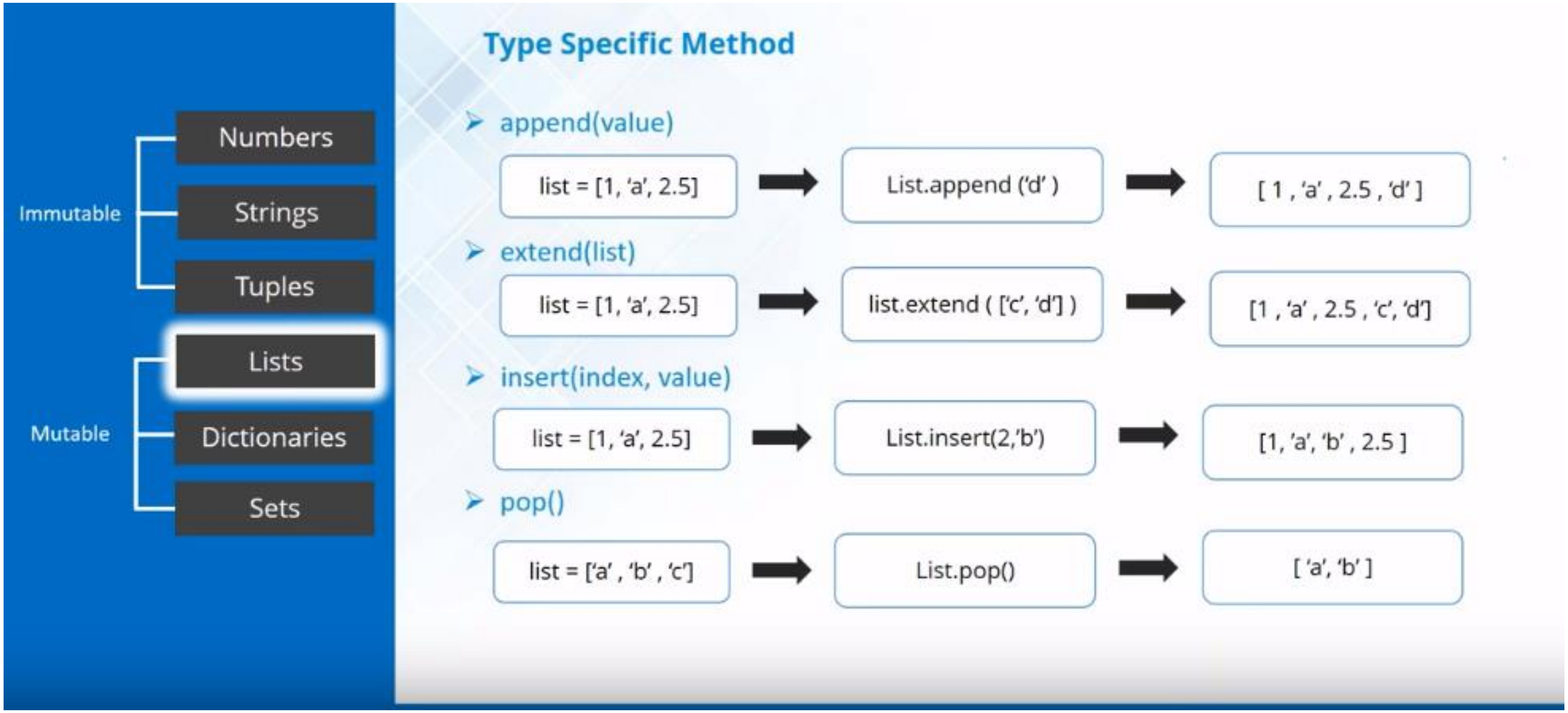
➢ Slicing

list = ['a' , 'b' , 'c' ,'d'] ➡ list[1:3] ➡ ('b' , 'c', 'd')

➢ Indexing

list = ['a' , 'b' , 'c'] ➡ list[0] ➡ 'a'

List Operations



Type Specific Method

- append(value)

list = [1, 'a', 2.5] → List.append ('d') → [1 , 'a' , 2.5 , 'd']

- extend(list)

list = [1, 'a', 2.5] → list.extend (['c', 'd']) → [1 , 'a' , 2.5 , 'c', 'd']

- insert(index, value)

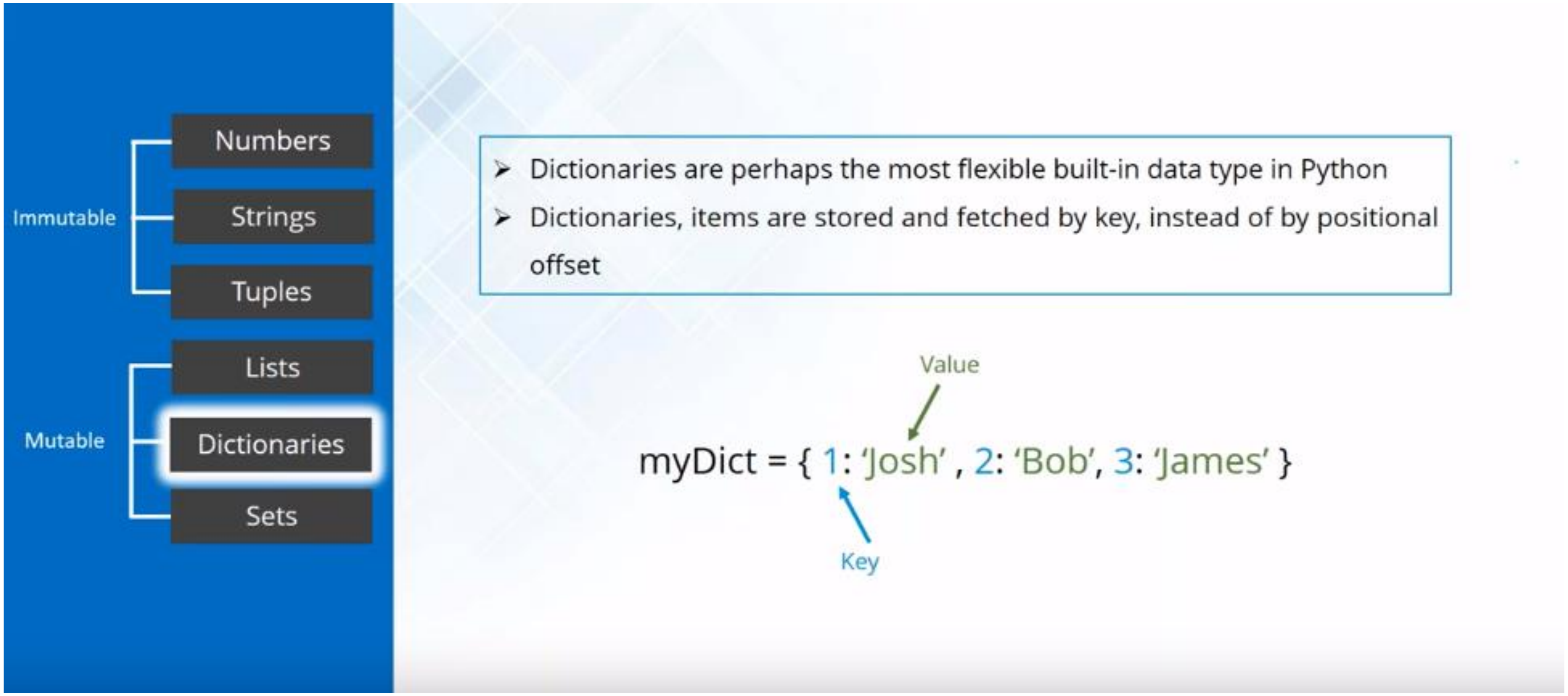
list = [1, 'a', 2.5] → List.insert(2,'b') → [1, 'a', 'b' , 2.5]

- pop()

list = ['a' , 'b' , 'c'] → List.pop() → ['a', 'b']

Datatype Cont…

▪

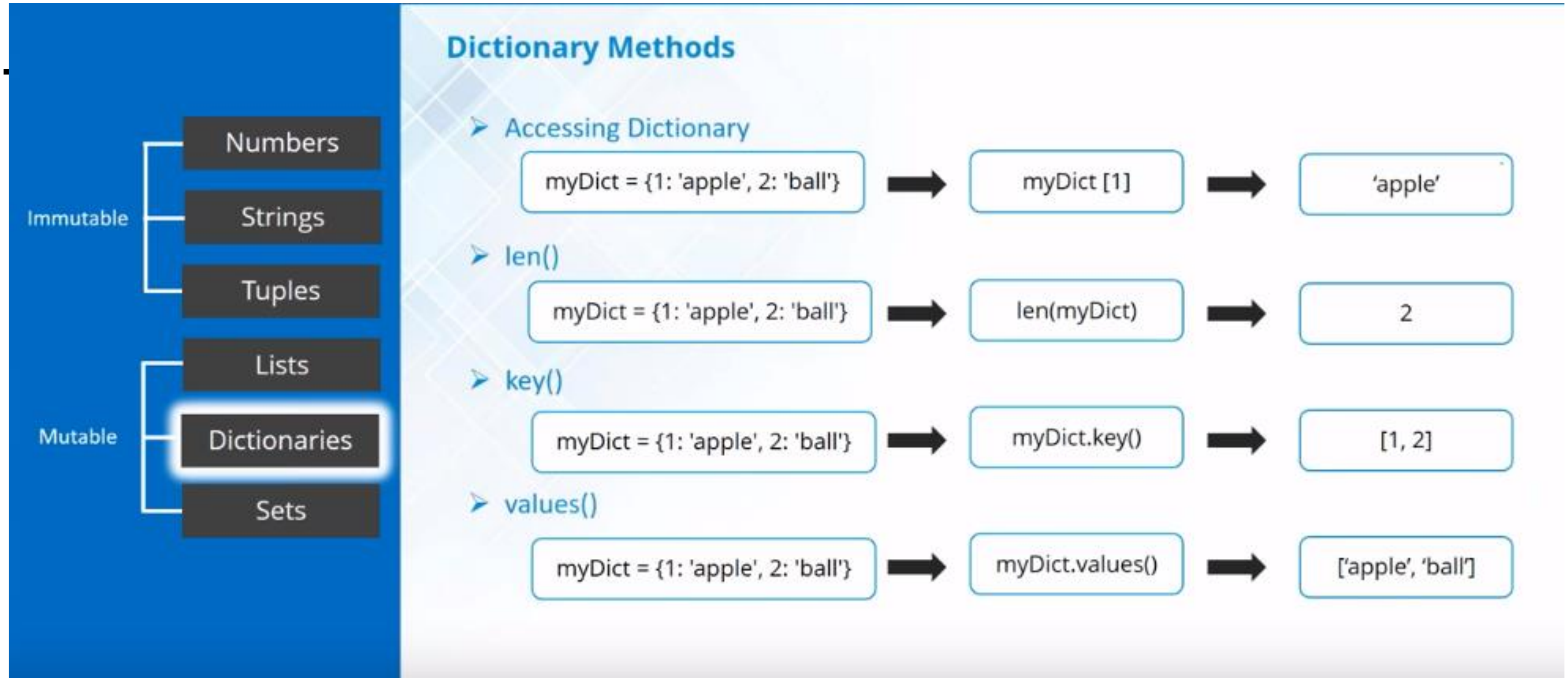


➢ Dictionaries are perhaps the most flexible built-in data type in Python

➢ Dictionaries, items are stored and fetched by key, instead of by positional offset

```
myDict = { 1: 'Josh' , 2: 'Bob', 3: 'James' }
```

Dictionary Methods



Dictionary Methods

- Accessing Dictionary

  myDict = {1: 'apple', 2: 'ball'} ➡ myDict [1] ➡ 'apple'

- len()

  myDict = {1: 'apple', 2: 'ball'} ➡ len(myDict) ➡ 2

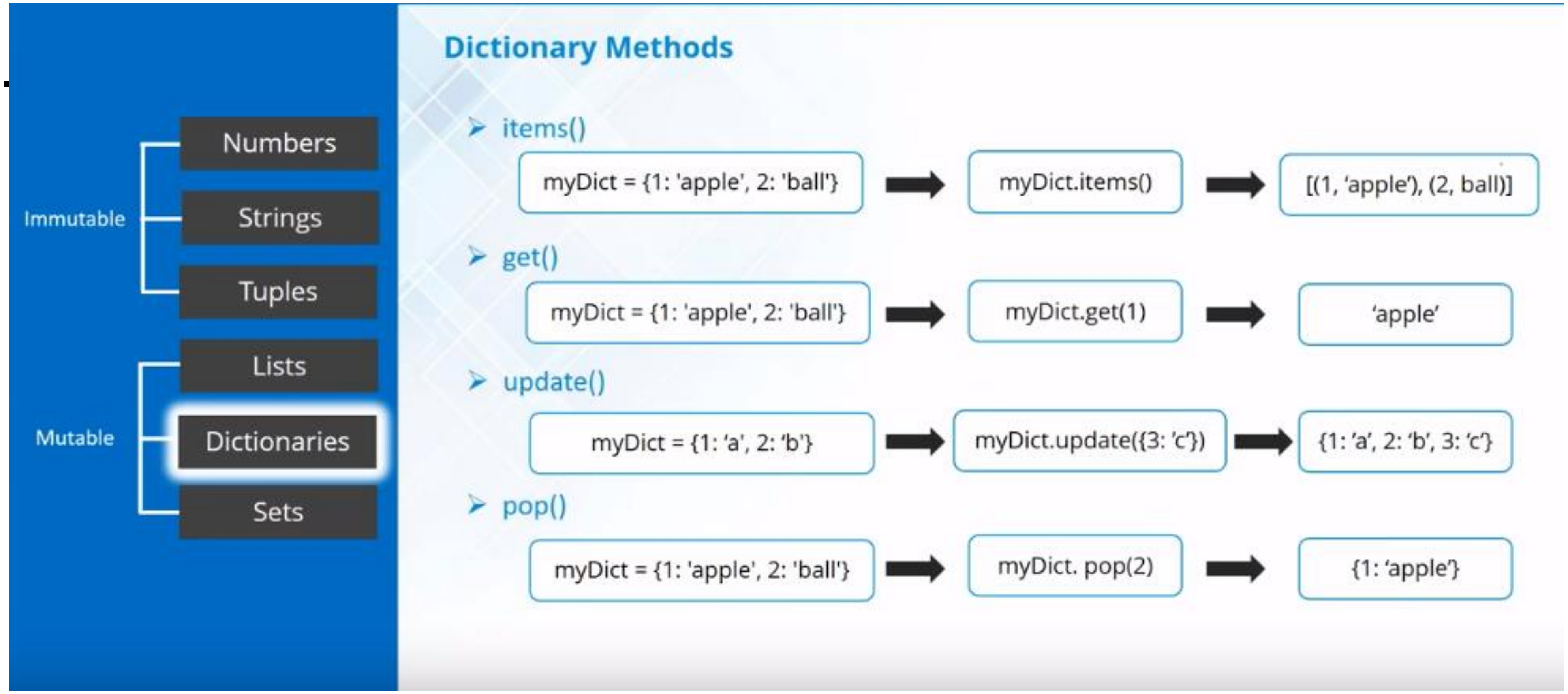
- key()

  myDict = {1: 'apple', 2: 'ball'} ➡ myDict.key() ➡ [1, 2]

- values()

  myDict = {1: 'apple', 2: 'ball'} ➡ myDict.values() ➡ ['apple', 'ball']

Dictionary Methods



Dictionary Methods

- items()

```
myDict = {1: 'apple', 2: 'ball'}  →  myDict.items()  →  [(1, 'apple'), (2, ball)]
```

- get()

```
myDict = {1: 'apple', 2: 'ball'}  →  myDict.get(1)  →  'apple'
```

- update()

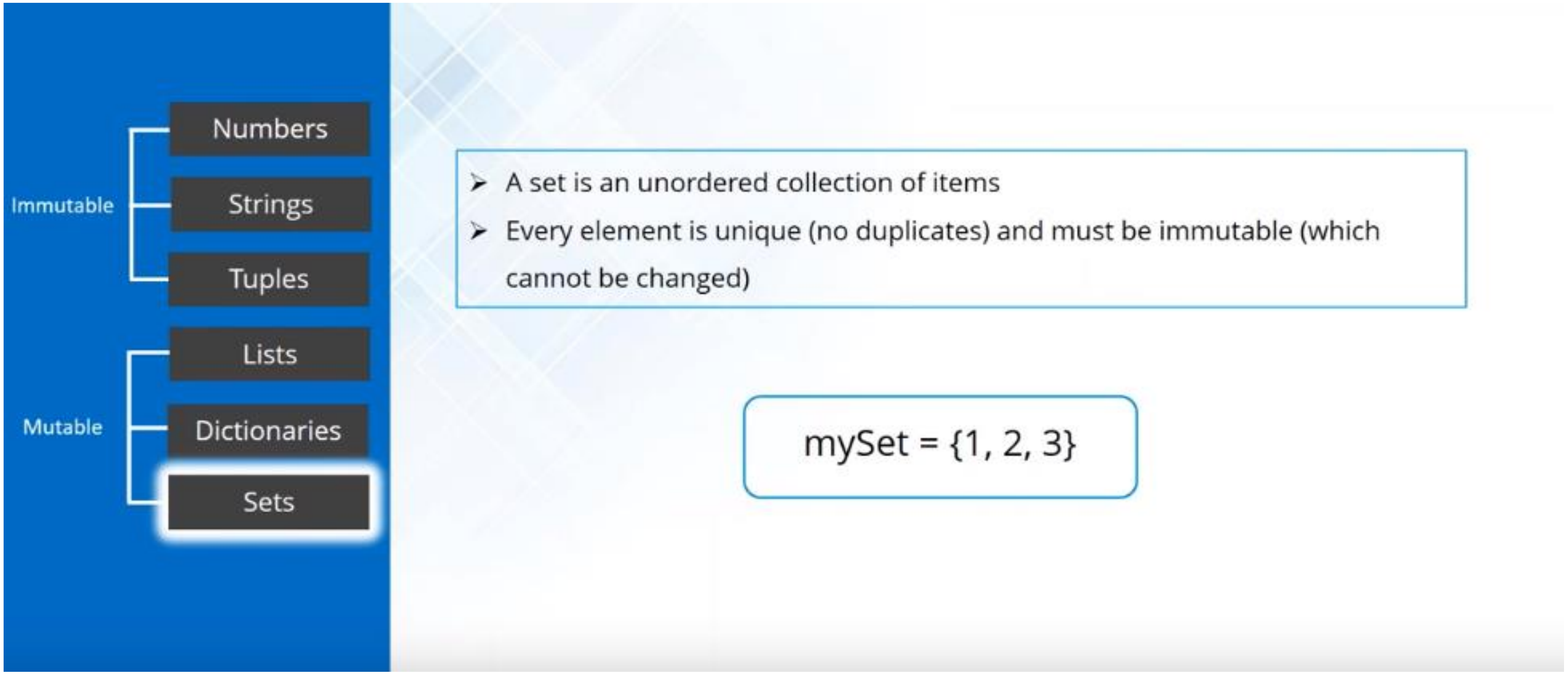
```
myDict = {1: 'a', 2: 'b'}  →  myDict.update({3: 'c'})  →  {1: 'a', 2: 'b', 3: 'c'}
```

- pop()

```
myDict = {1: 'apple', 2: 'ball'}  →  myDict. pop(2)  →  {1: 'apple'}
```

Datatype Cont…

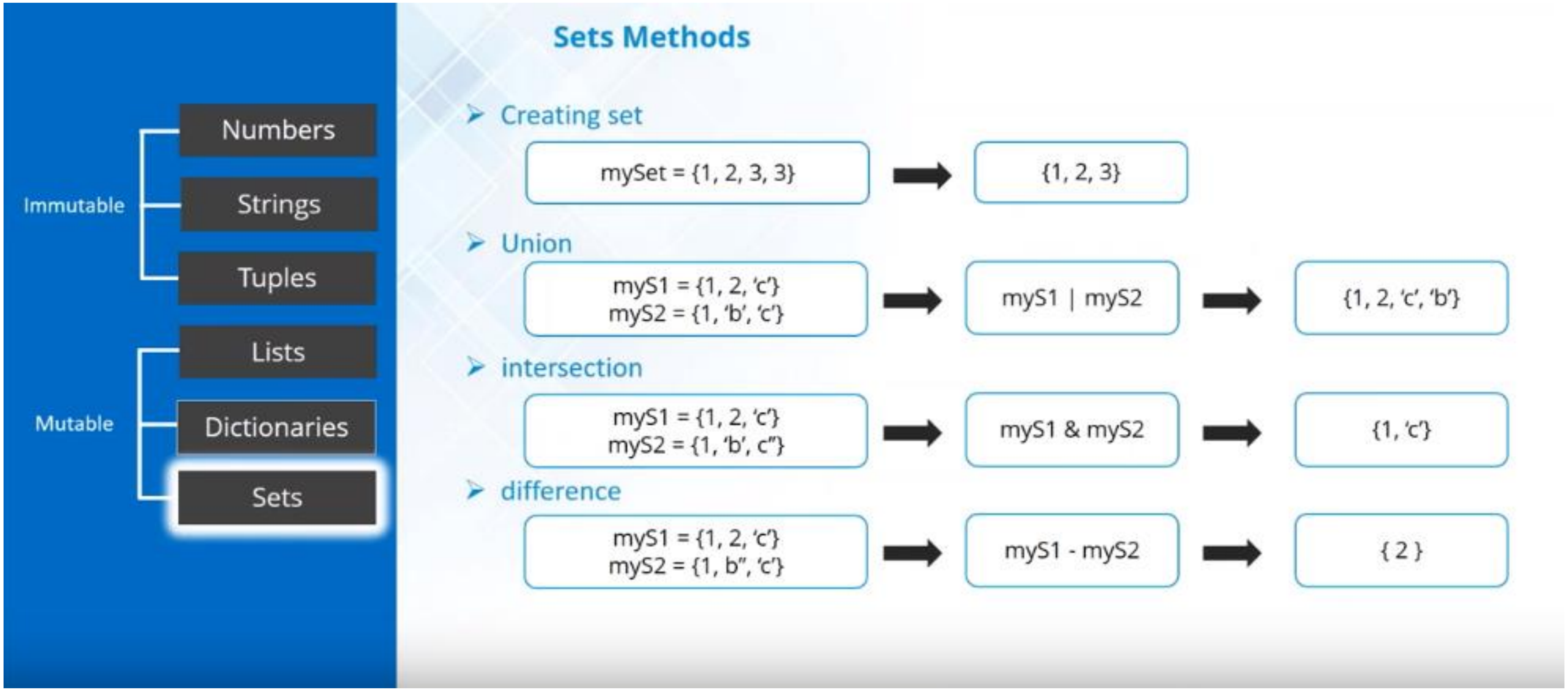
▪



- A set is an unordered collection of items
- Every element is unique (no duplicates) and must be immutable (which cannot be changed)

```
mySet = {1, 2, 3}
```

Set Methods



Sets Methods

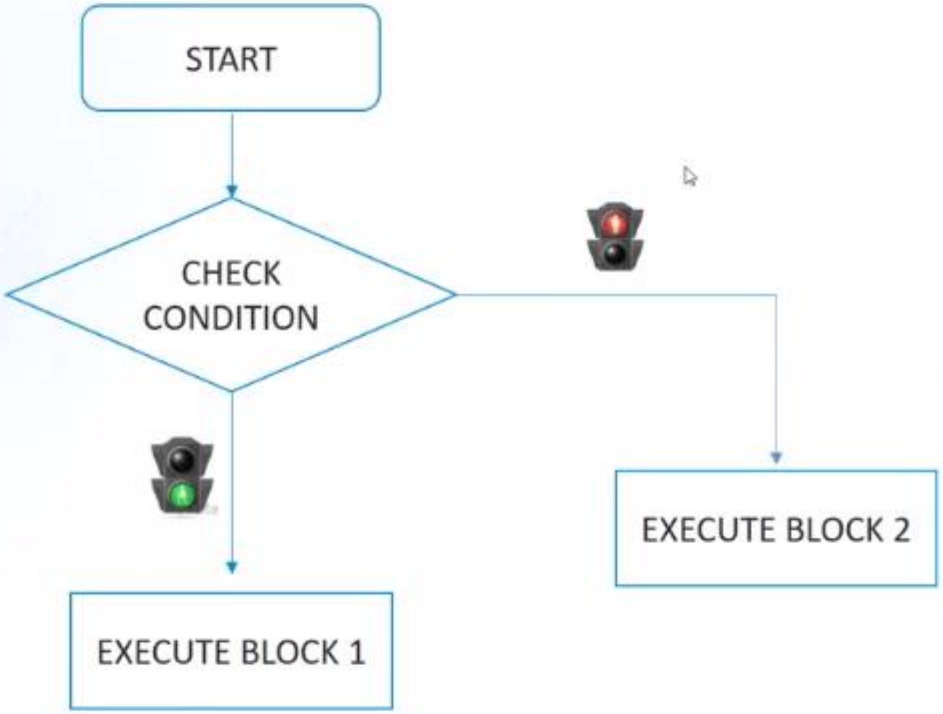
- Creating set
  - mySet = {1, 2, 3, 3} → {1, 2, 3}
- Union
  - mySI = {1, 2, 'c'} mySZ = {1, 'b', 'c'} → myS1 | myS2 → {1, 2, 'c', 'b'}
- intersection
  - myS1 = {1, 2, 'c'} myS2 = {1, 'b', c"} → myS1 & myS2 → {1, 'c'}
- difference
  - myS1 = {1, 2, 'c'} myS2 = {1, b", 'c'} → myS1 - myS2 → { 2 }

Flow Control Cont…

1. If statement

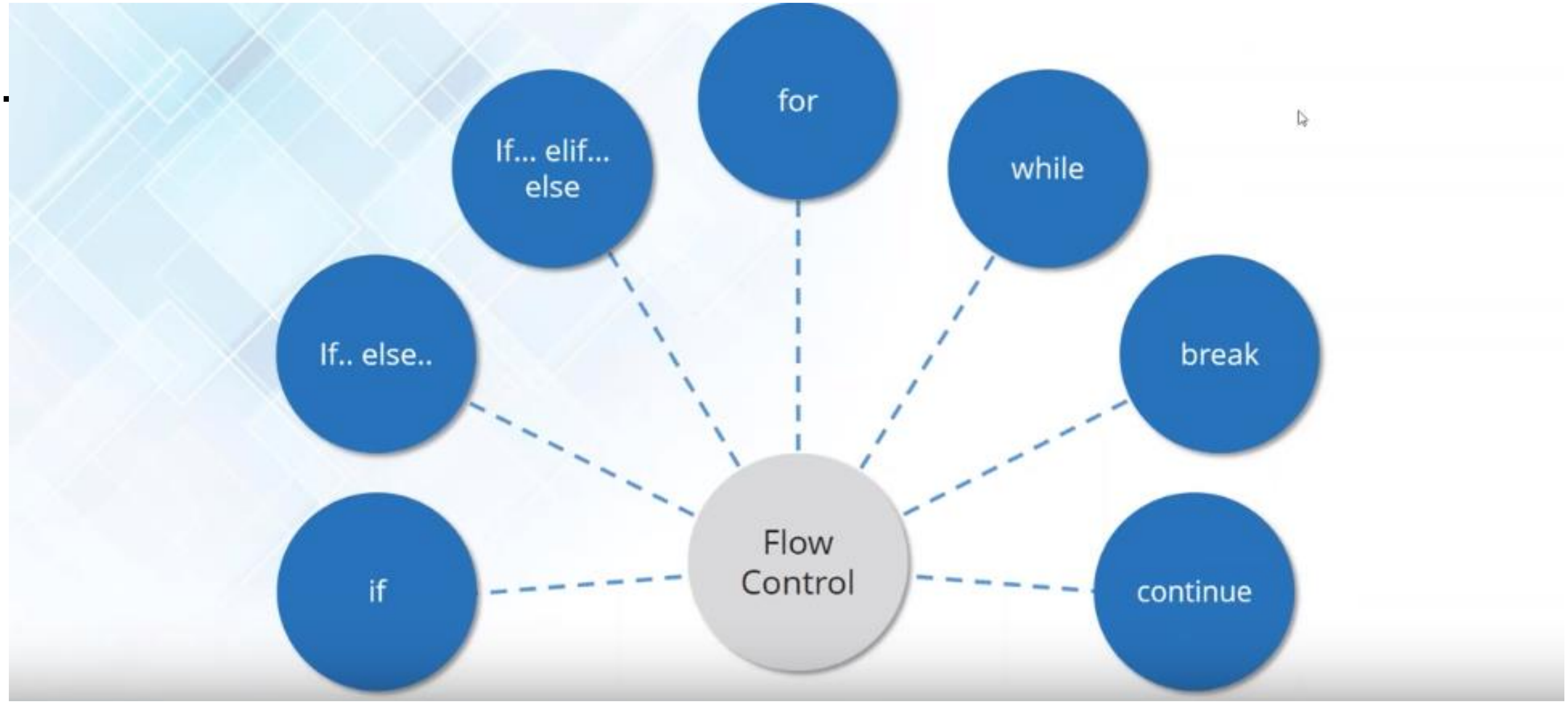
Syntax:

```
if (condition):
    statements 1 ...
else:
    statements 2 ...
```



Flow Control

▪

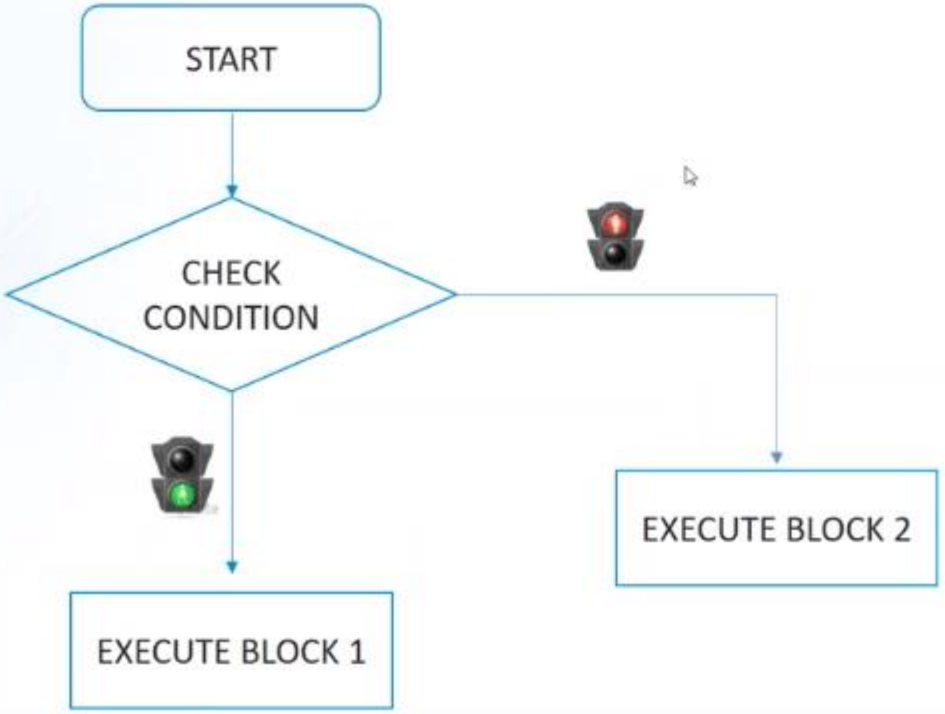


Flow Control Cont…

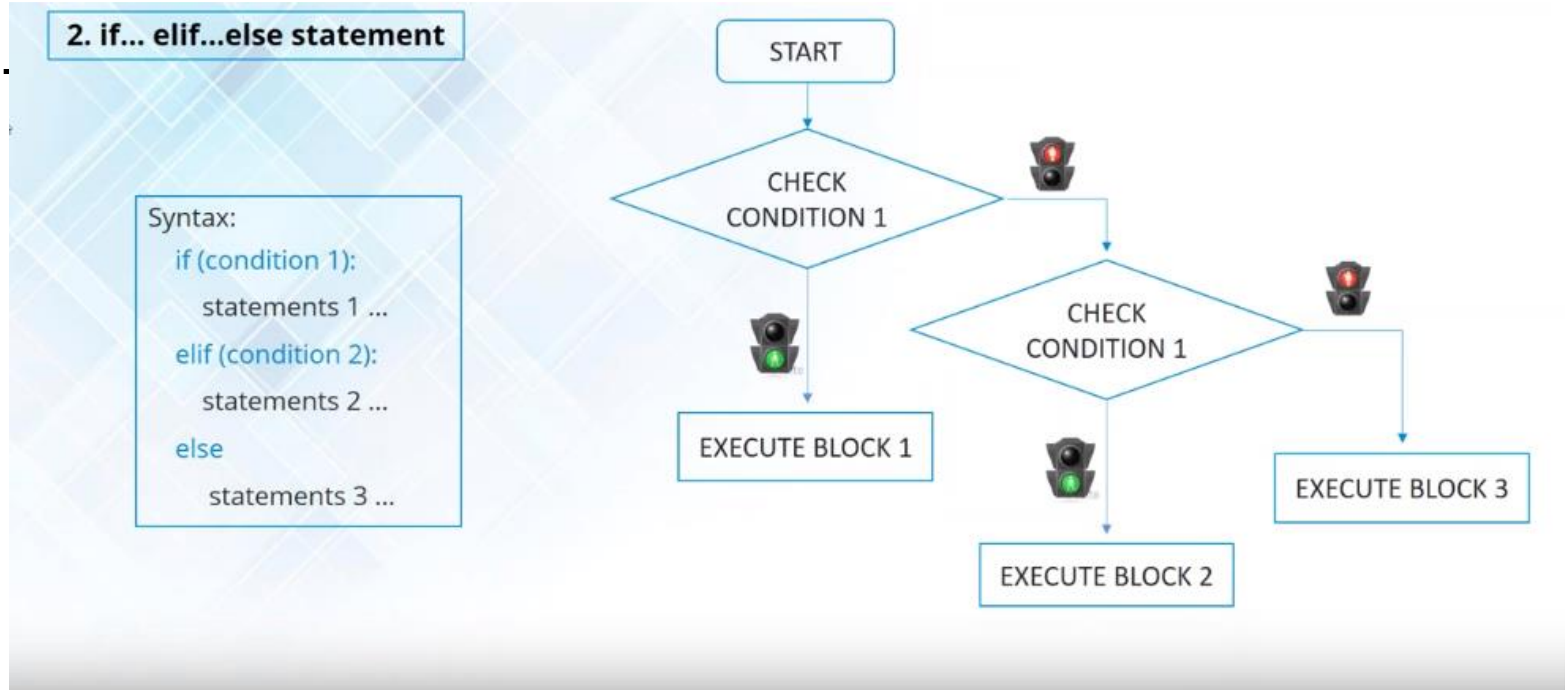
1. If statement

Syntax:

```
if (condition):
    statements 1 ...
else:
    statements 2 ...
```



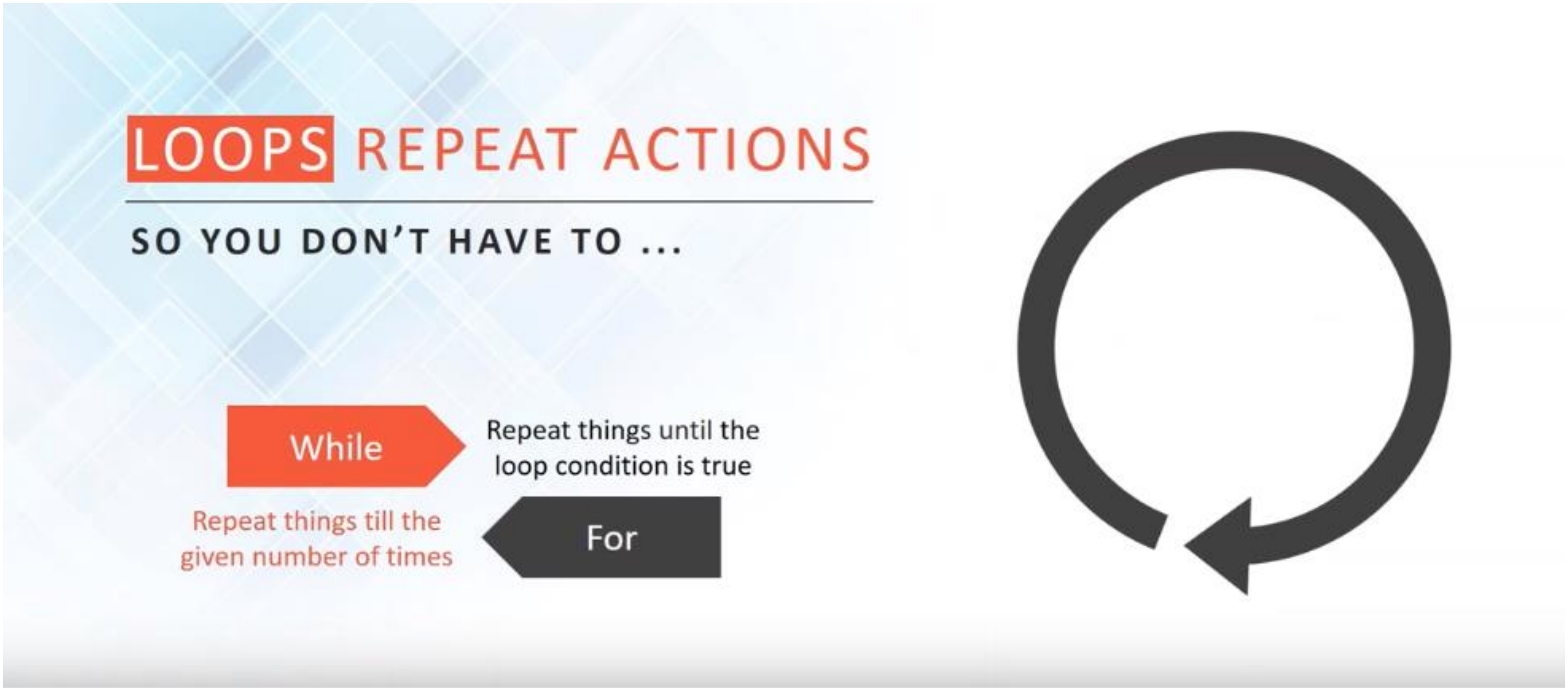
Flow Control Cont…



2. if... elif...else statement

```
Syntax:
   if (condition 1):
      statements 1 ...
   elif (condition 2):
      statements 2 ...
   else
      statements 3 ...
```

Flow Control Cont…



LOOPS REPEAT ACTIONS

SO YOU DON'T HAVE TO …

While

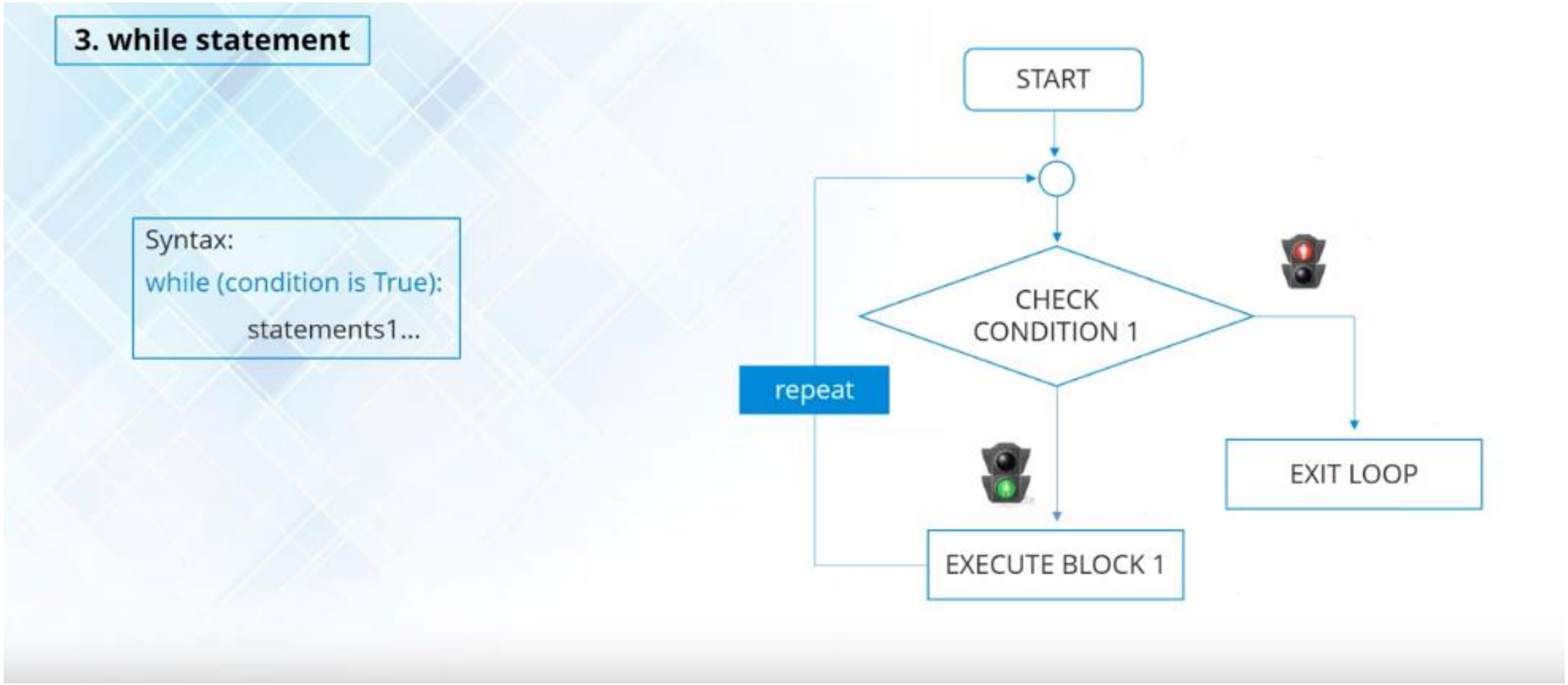
Repeat things until the loop condition is true

For

Repeat things till the given number of times

Flow Control Cont…

▪



3. while statement

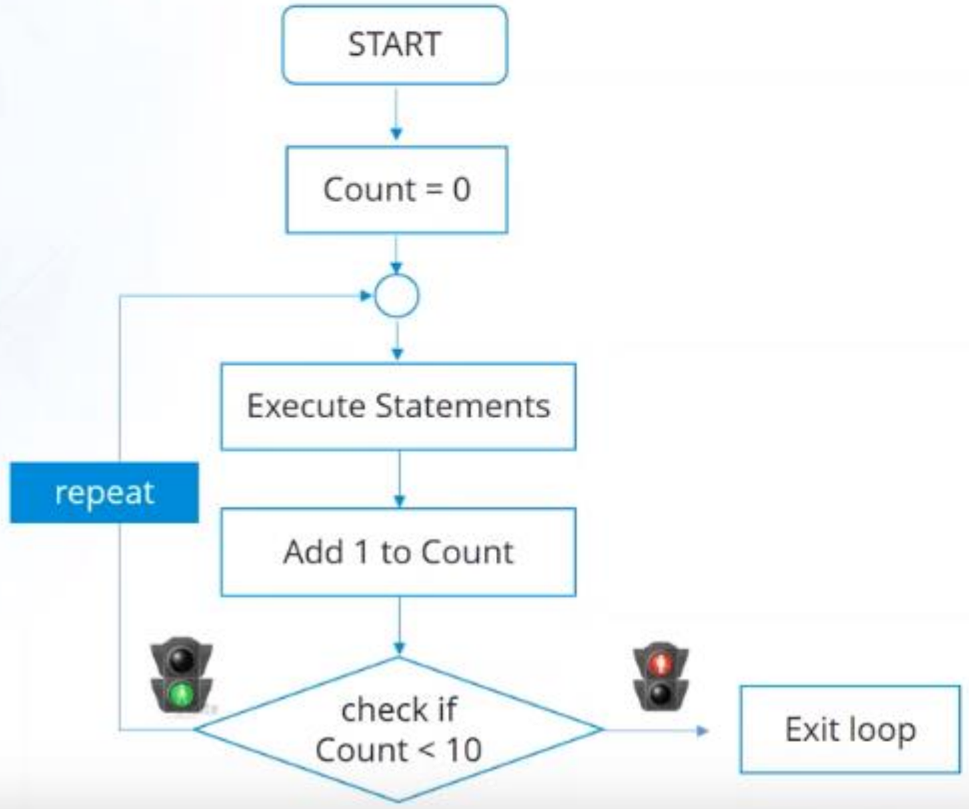
Syntax:

```
while (condition is True):
        statements1...
```

Flow Control Cont…

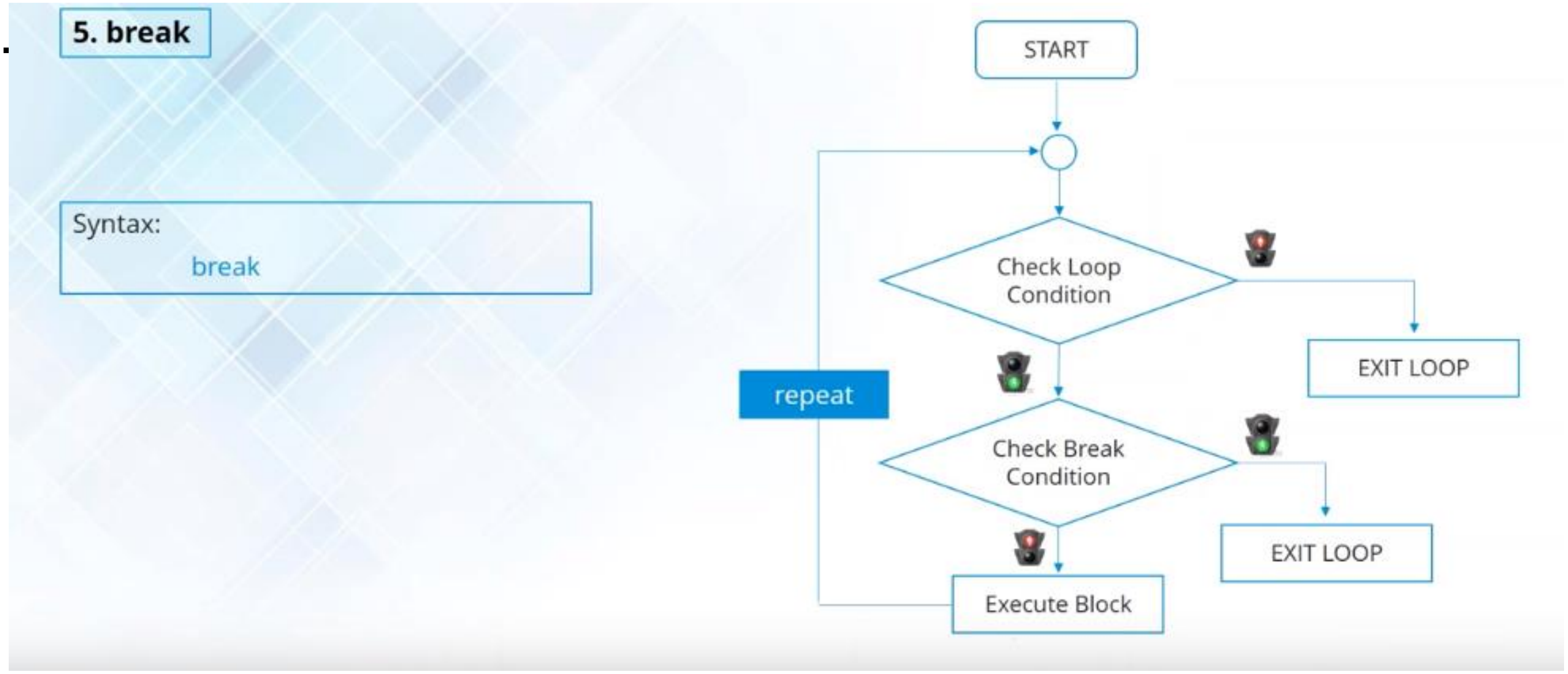
4. for statement

Syntax:
for iterator name in iterating sequence:
execute statements…



Flow Control Cont…

▪


5. break

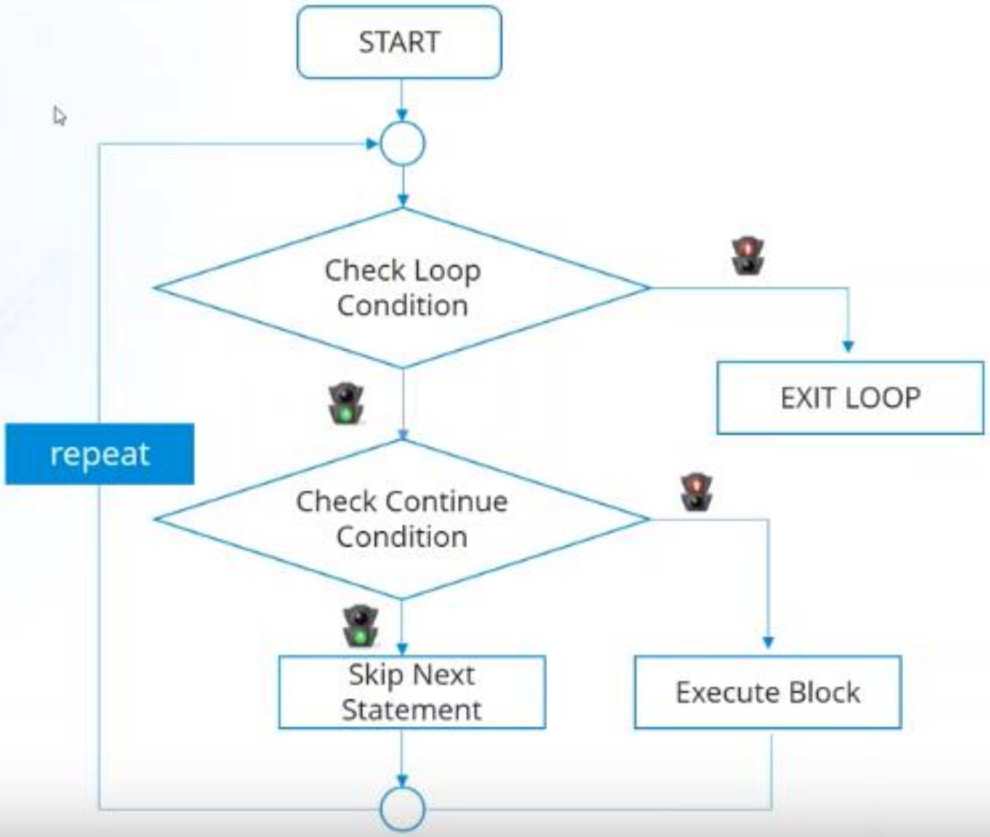
Syntax:
```
        break
```

Flow Control Cont…

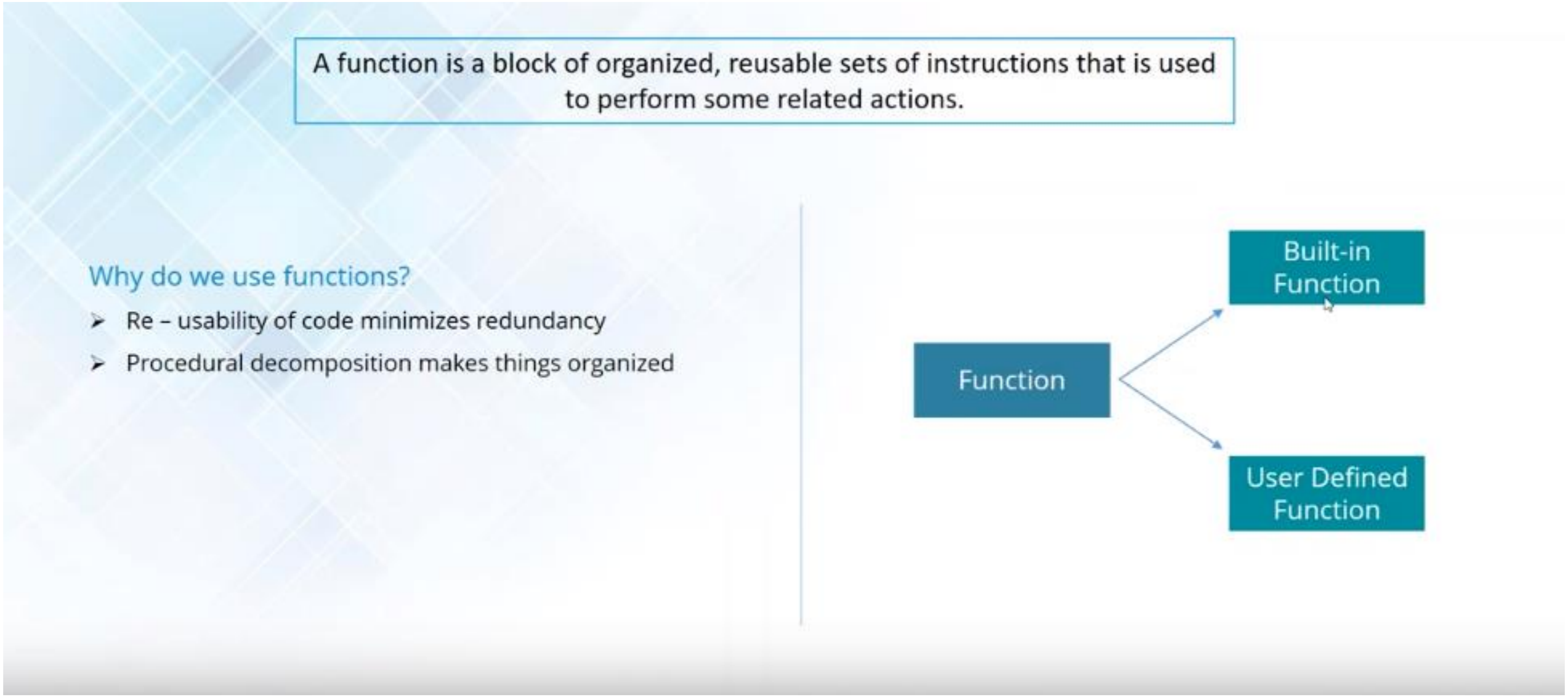
6. continue

Syntax:

```
continue
```



Function



A function is a block of organized, reusable sets of instructions that is used to perform some related actions.

Why do we use functions?

- Re – usability of code minimizes redundancy
- Procedural decomposition makes things organized

Function Cont…

- A Function is set of instruction that is used to perform a single repeated operation
- Functions are:

  > Organized

  > Re-usable

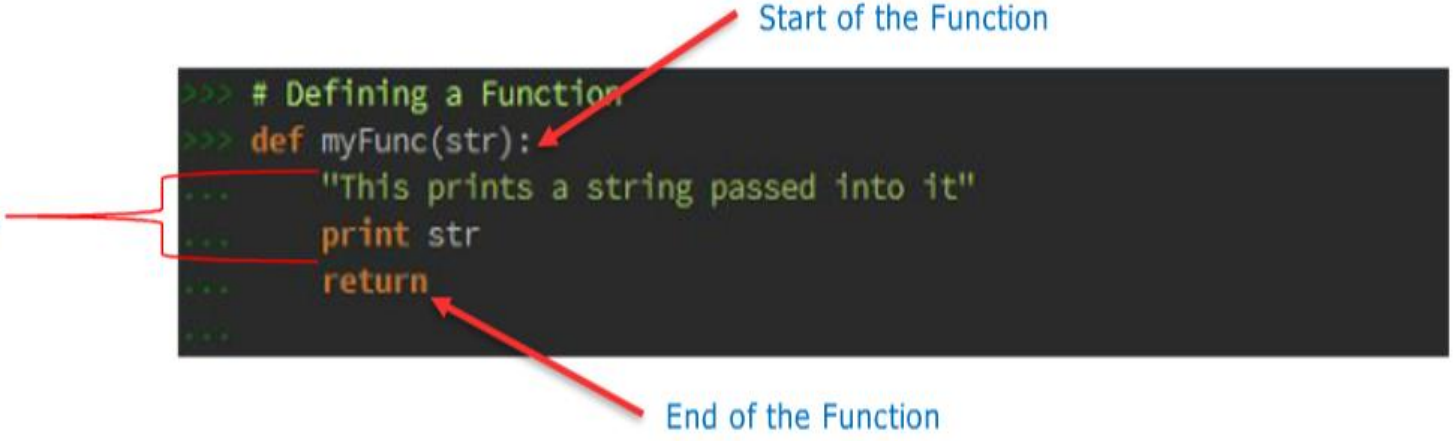
  > Modularized
- Python has built in functions like print(),input(),int(),len(), etc. but you can also create your own functions
- There functions are called user-defined Functions

Syntax:

```
def name(param1, param2 …….paramN):
        Statements…
        return [ expression ]
```

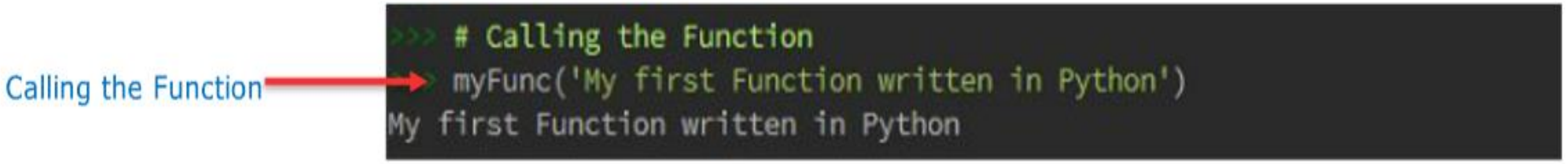
Defining and calling a Function:

Defining a Function



```
>>> # Defining a Function
>>> def myFunc(str):
...     "This prints a string passed into it"
...     print str
...     return
...
```

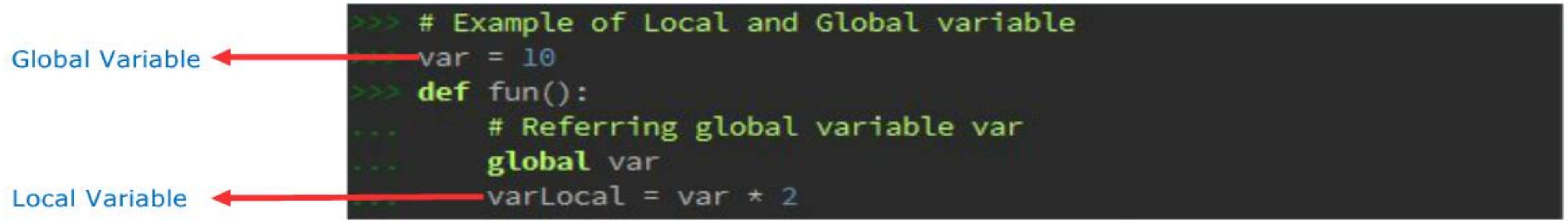
Calling the Function



```
>>> # Calling the Function
>>> myFunc('My first Function written in Python')
My first Function written in Python
```

Function (Local and Global Variable):

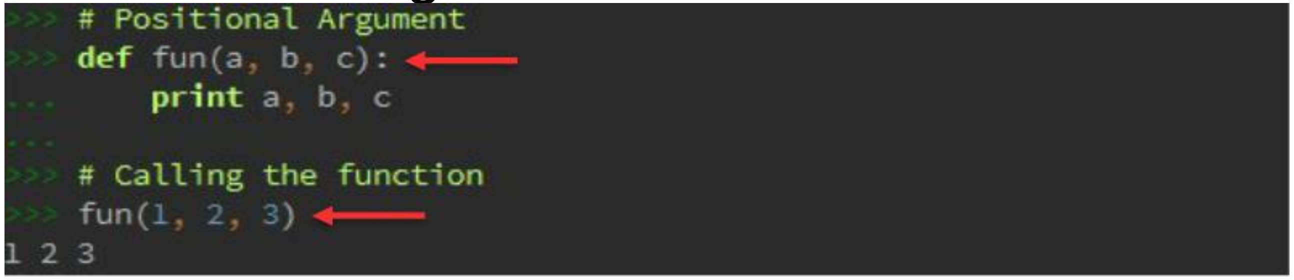
- Local Variables:
  - ➢ The variable defined within the function has a local scope and hence they are called local variables.
  - ➢ Local scope means they can be accessed within the function only
  - ➢ They appear when the function is called and disappear when the function exits
- Global Variables:
  - ➢ The variable defined outside the function has a global scope and hence they are called global variables.
  - ➢ Global scope means they can be accessed within the function as well as outside the function
  - ➢ The value of the global variable can be used by referring as global inside a function



```
>>> # Example of Local and Global variable
>>> var = 10
>>> def fun():
...     # Referring global variable var
...     global var
...     varLocal = var * 2
```

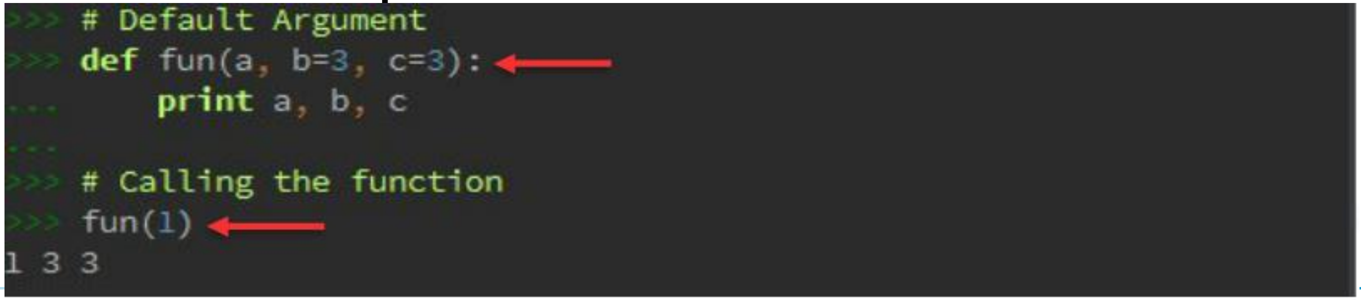
Various Form of Function Arguments:

- It is possible to define a function with a variable number of argument
.
- You can call a function by using the following arguments:
- Position Argument -> Arguments passed should match the function parameter from left to right



```
>>> # Positional Argument
>>> def fun(a, b, c):
...     print a, b, c
...
>>> # Calling the function
>>> fun(1, 2, 3)
1 2 3
```

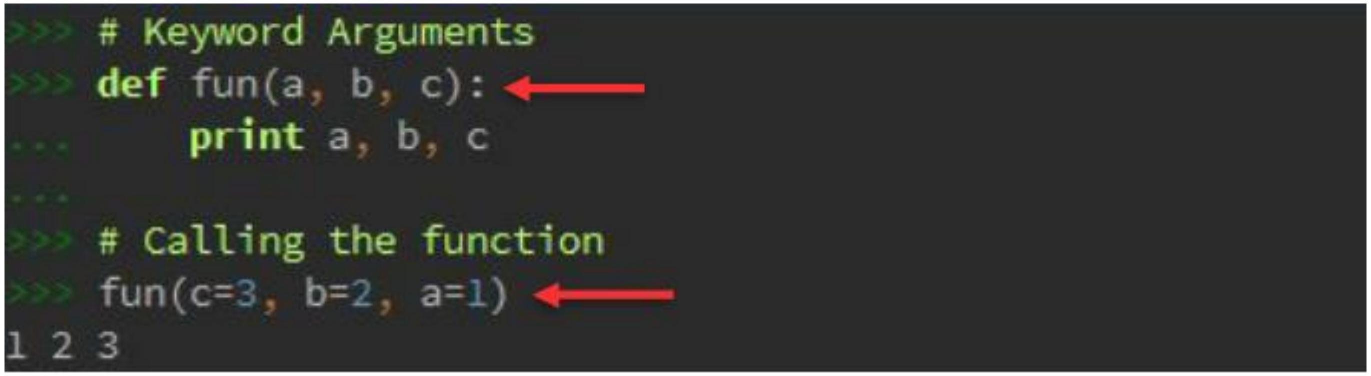
- Default Argument Values -> We can assign default value for arguments to receive if the call passes too few values



```
>>> # Default Argument
>>> def fun(a, b=3, c=3):
...     print a, b, c
...
>>> # Calling the function
>>> fun(1)
1 3 3
```

Various Form of Function Arguments:

- Keyword Argument -> We can call a function by specifying the keyword argument in the form argument name = value



```
>>> # Keyword Arguments
>>> def fun(a, b, c):
...     print a, b, c
...
>>> # Calling the function
>>> fun(c=3, b=2, a=1)
1 2 3
```

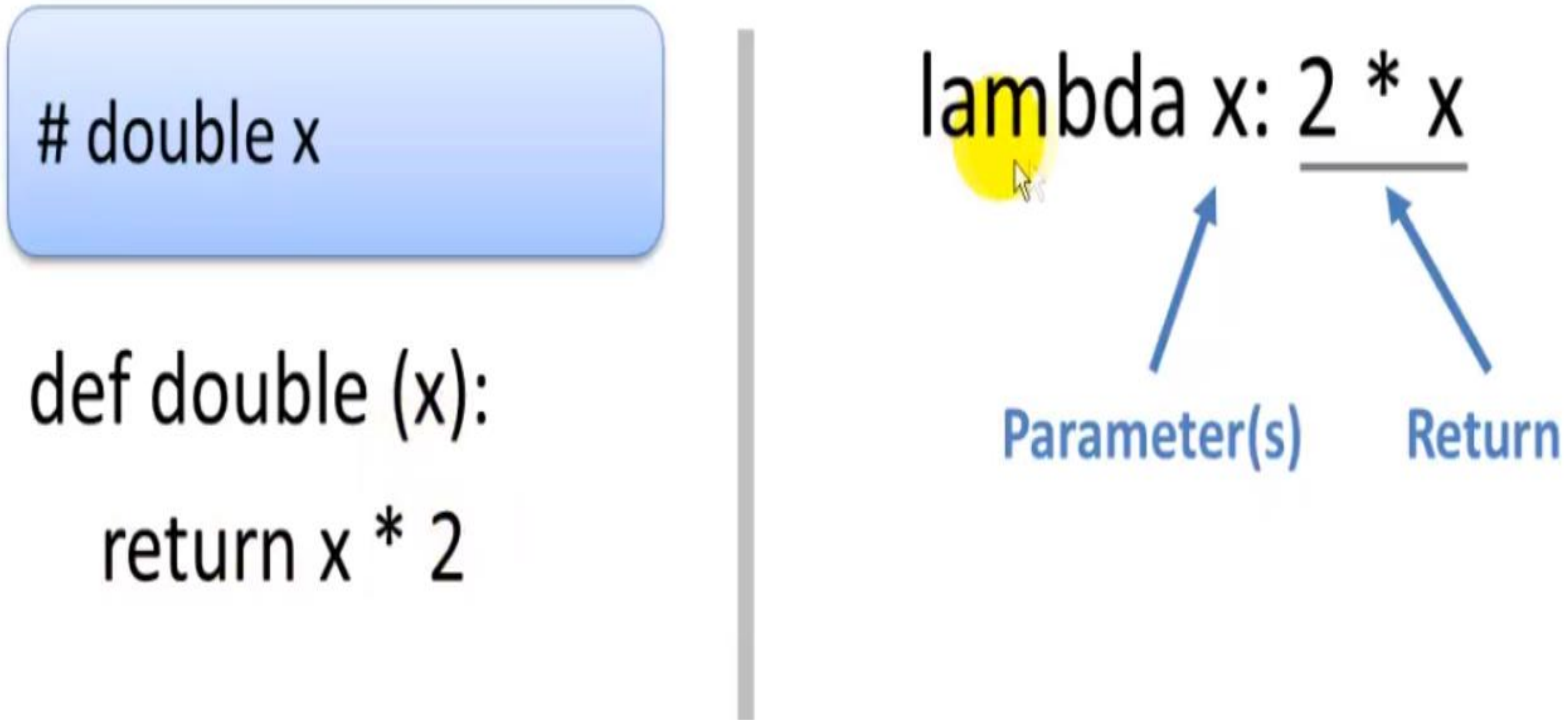
Various Form of Function Arguments:

- Arbitrary Argument List -> You may need to process a function for more arguments than you specified while defining the function.
- These arguments are not named in the function definition. They are used when you need to pass more arguments that you have specified while defining the function. To add an arbitrary argument in the function definition , start the variable name with * or **
- `>>> def fun(*arr)`
- `>>> def fun(**arr)`

Lambda Function

- A simple one line function
- Do not use def or return keywords. These are implicit

Lambda Example:



```
# double x

def double (x):
    return x * 2
```

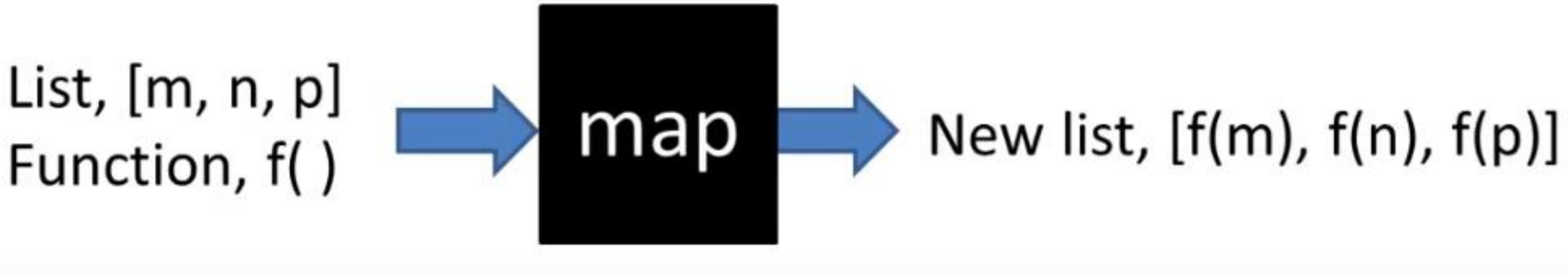
```
lambda x: 2 * x
```

Map Example:

- Apply same function to each element of a sequence

.

- Return the modified list



Map Example:

```
# prints [16, 9, 4, 1]
```

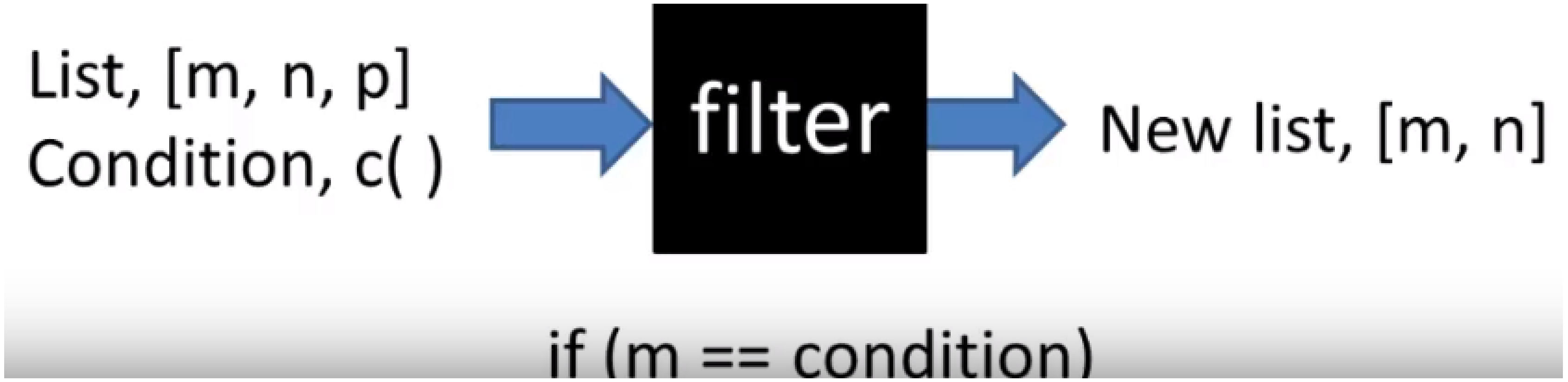
```
def square (lst1):
    lst2 = []
    for num in lst1:
        lst2.append(num ** 2)
    return lst2

print square([4,3,2,1])
```

```
n = [4, 3, 2, 1]
print (list(map(lambda x: x**2, n)))
```

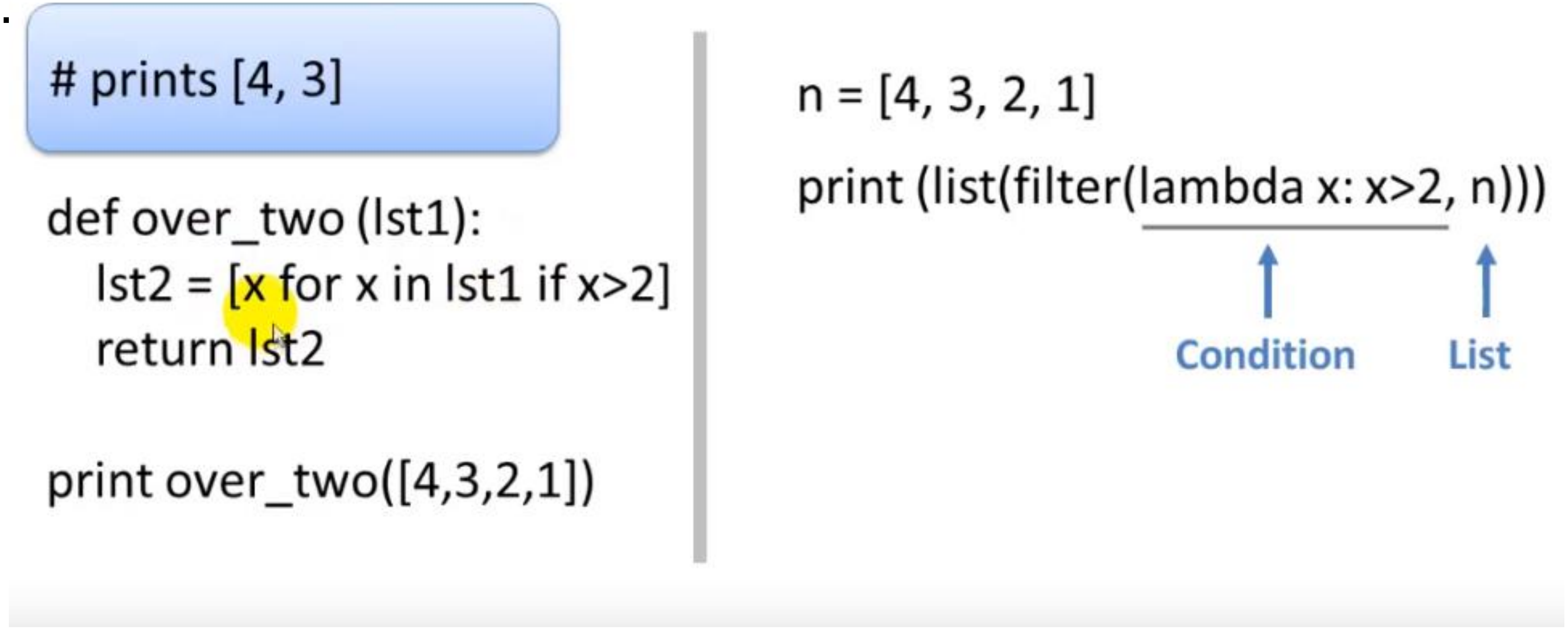
Function   List

Filter Function:

- Filter items out of a sequence
- Return the filtered list



Filter Function:

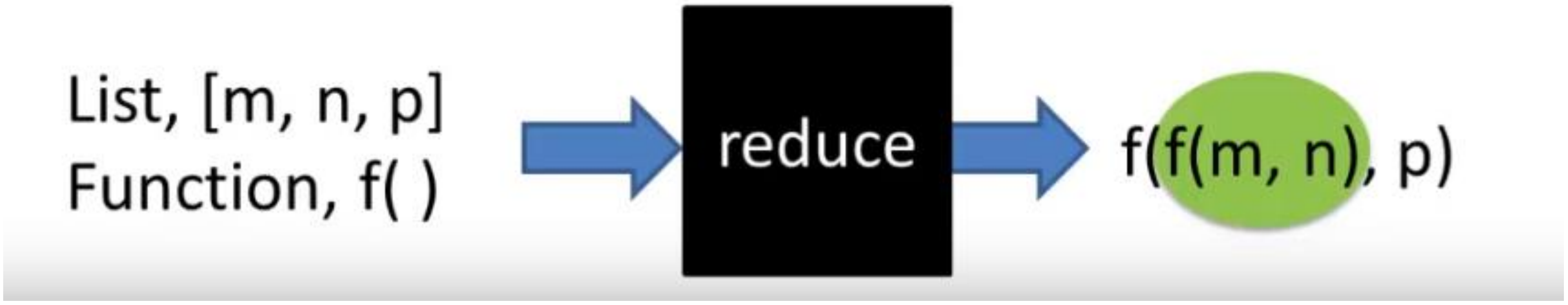


```
# prints [4, 3]

def over_two (lst1):
    lst2 = [x for x in lst1 if x>2]
    return lst2

print over_two([4,3,2,1])
```

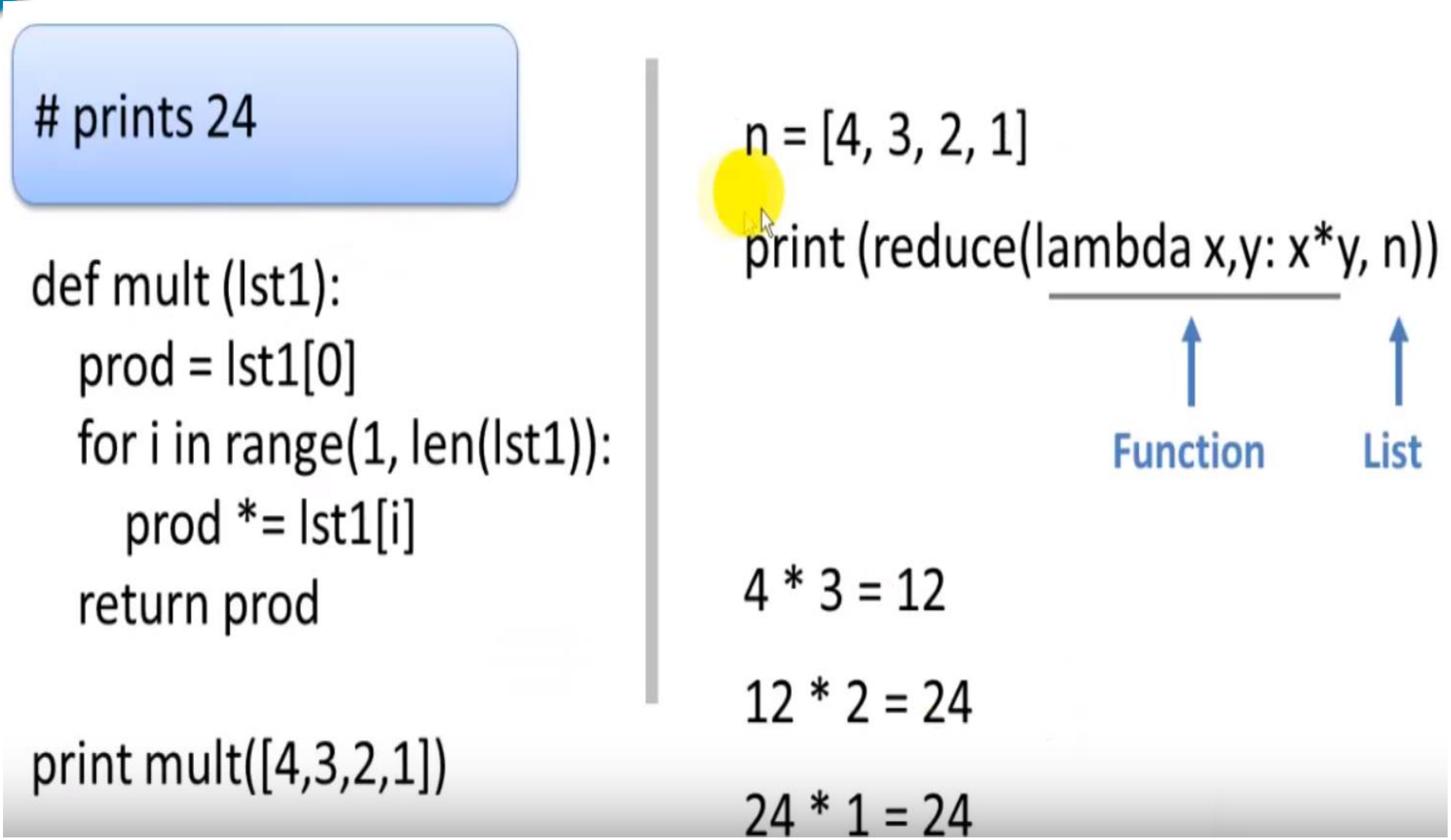
```
n = [4, 3, 2, 1]

print (list(filter(lambda x: x>2, n)))
```

Reduce Function:

- Applies same operation to items of a sequence
- Uses result of operation as first param of next operation
- Returns an item, not a list



Reduce Function:



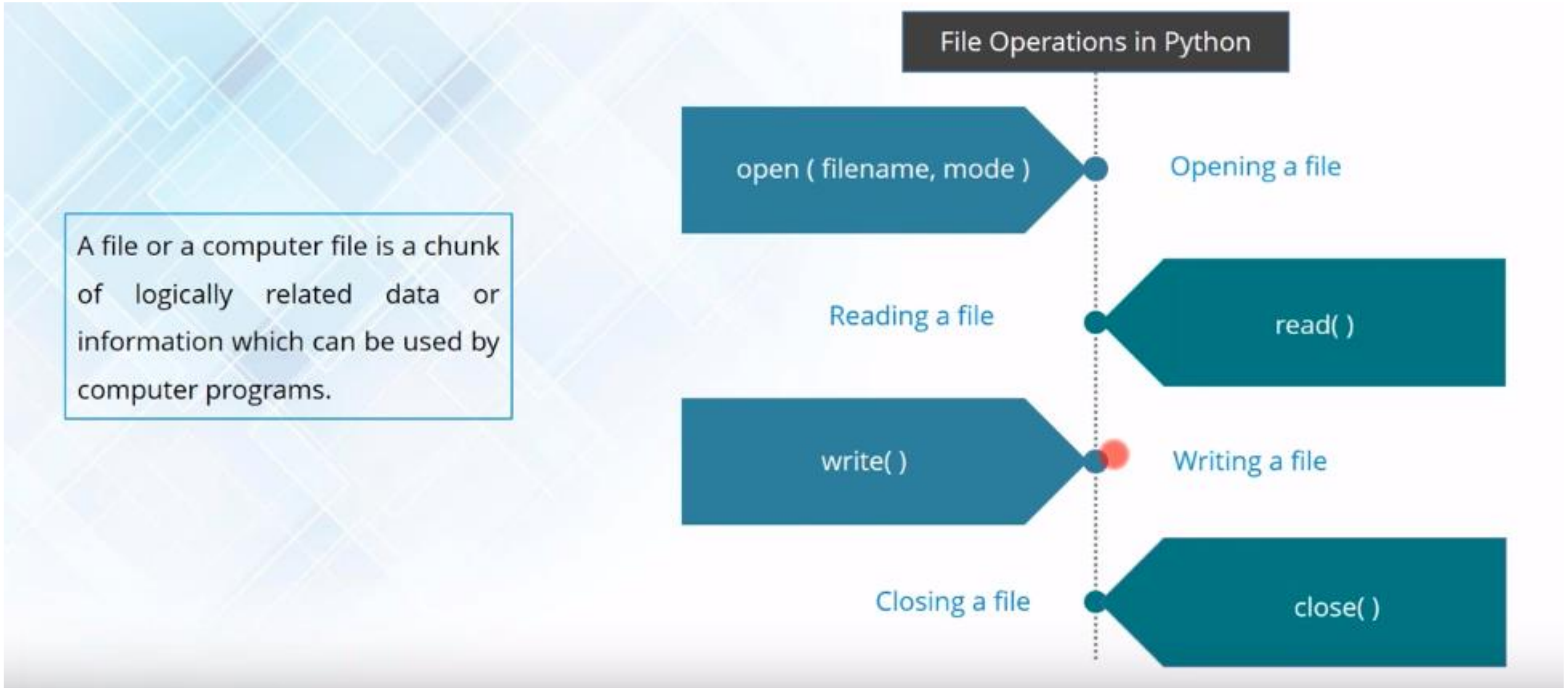
```
# prints 24

def mult (lst1):
    prod = lst1[0]
    for i in range(1, len(lst1)):
        prod *= lst1[i]
    return prod

print mult([4,3,2,1])
```

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

print (reduce(lambda x,y: x*y, n))
```

```
4 * 3 = 12

12 * 2 = 24

24 * 1 = 24
```

File Handling & I/O Method

▪



A file or a computer file is a chunk of logically related data or information which can be used by computer programs.

Mode of opening a file

Modes	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer will be at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer will be at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

The File Attribute:

▪

Attribute	Description
file.closed	Returns true if file is closed, false otherwise
file.mode	Returns access mode with which file was opened
file.name	Returns name of the file
file.softspace	Returns false if space explicitly required with print, true otherwise

Common File Methods:

→ write()

- » The write() method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.
- » Syntax: fileObject.write(string);

→ read()

- » The read() method reads a string from an open file. It is important to note that Python strings can have binary data and not just text.
- » Syntax: fileObject.read([count]);

→ close()

- » The close() method of a file object flushes any unwritten information and closes the file object, after which no more writing can be done.
- » Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.
- » Syntax: fileObject.close();

Common File Methods:

→ write()

» The write() method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.
» Syntax: fileObject.write(string);

→ read()

» The read() method reads a string from an open file. It is important to note that Python strings can have binary data and not just text.
» Syntax: fileObject.read([count]);

→ close()

» The close() method of a file object flushes any unwritten information and closes the file object, after which no more writing can be done.
» Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.
» Syntax: fileObject.close();

Errors and Exception Handling :

- What is an Exception?
    - An Exception is an error that happens during execution of a program. When that error occurs , Python generates an exception that can be handled, which avoids your program to crash
- Why use Exceptions?
    - Exception are convenient in many ways for handling errors and special conditions in program. When you think that you have a code which can produce an error then you can use an exception

- You can raise an exception in your own program by using the raise exception statement.
- Raising an exception breaks current code execution and returns the exception back until it is handled.

Common Exception Errors

- IOError

  >> If the file cannot be opened

- ImportError

  >> If python cannot find a module

- ValueError

  >> Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value

- KeyboardInterrupt

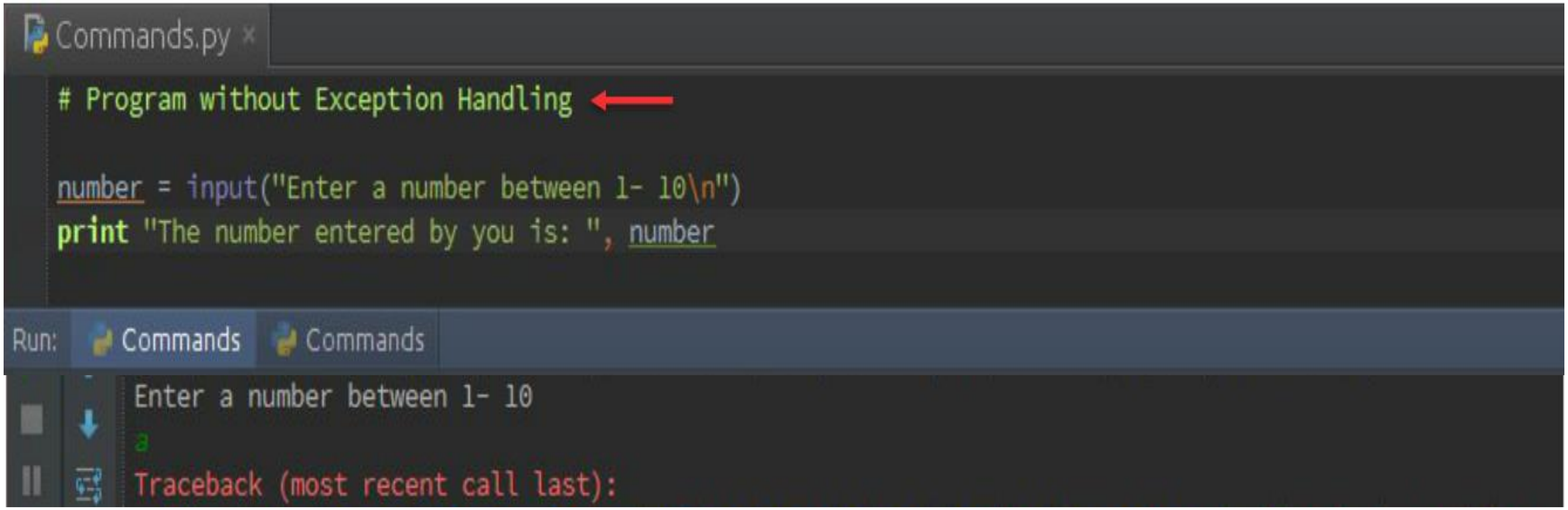
  >> Raised when use hits the interrupt key (normally ctrl+c or delete)

- EOFError

  >> Raised when one of the built-in functions(input() or raw input()) hits an end-of-file condition(EOF) without reading any data.

Common Exception Errors

- Program without Error Handling

▪ 

Commands.py

```
# Program without Exception Handling

number = input("Enter a number between 1- 10\n")
print "The number entered by you is: ", number
```

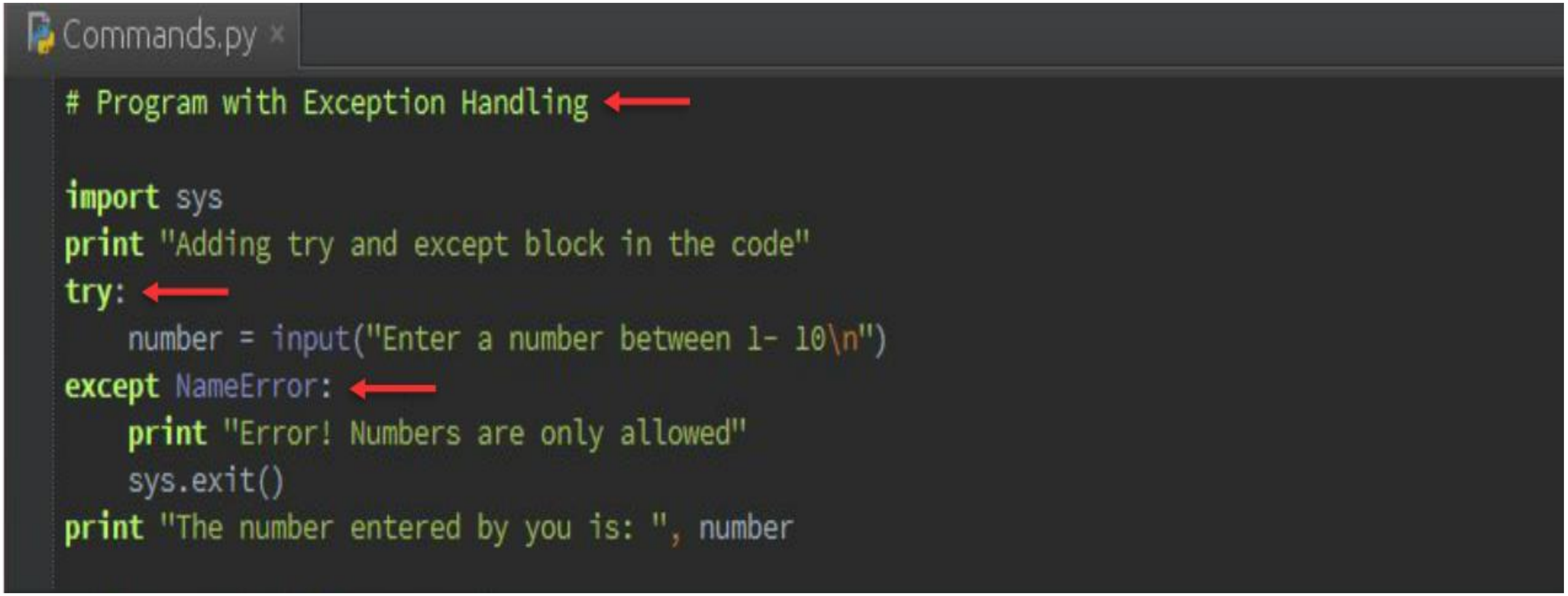
Run: Commands Commands

```
Enter a number between 1- 10
a
Traceback (most recent call last):
```

Common Exception Errors

- Program with Error Handling

▪



```
# Program with Exception Handling

import sys
print "Adding try and except block in the code"
try:
    number = input("Enter a number between 1- 10\n")
except NameError:
    print "Error! Numbers are only allowed"
    sys.exit()
print "The number entered by you is: ", number
```

try..except..else Clause

- The else clause in a try, except statement must follow all except clauses
- It is useful for code that must be executed if the try clause does not raise an exception
- try:
  - data = something_that_can_go_wrong
- except IOError:
  - Handle the exception error
- else:
  - Doing_different_exception_handlining

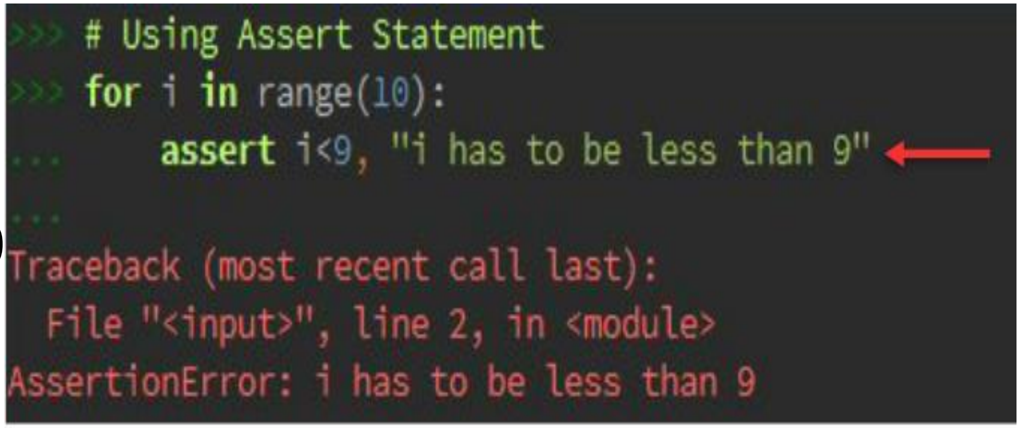
Note 1: Exceptions in the else clause are not handled by the preceding except clauses

Note 2: Make sure that the else clause is executed before the finally block

Assert Statement:

- The Assert statement is intended for debugging statements
- It raises an exception as soon as the condition is false
- The caller gets an exception which will go into stderr or syslog
    - assert <some_test_> ,<message>
- The line above can be "read" as : if <some_test> evaluates to False , an exception is raised and <message> will be output
    - >>> if not condition:
        - raise AsstionError()

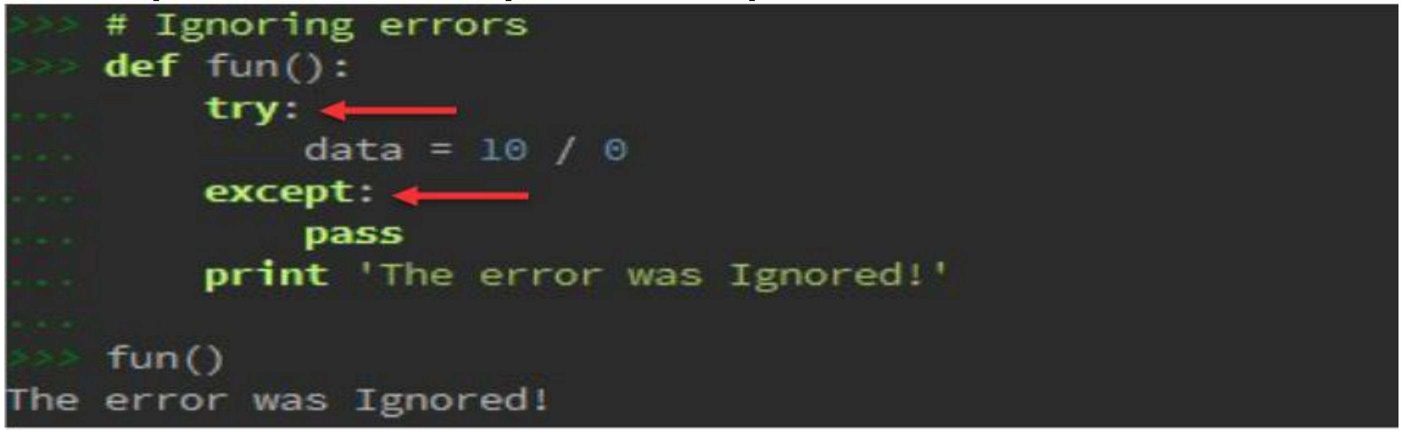
- **Example**:
  We have a loop which can result wrong if the value of i>=9 Hence we can use assert statement here



```
>>> # Using Assert Statement
>>> for i in range(10):
...     assert i<9, "i has to be less than 9"
...
Traceback (most recent call last):
  File "<input>", line 2, in <module>
AssertionError: i has to be less than 9
```

Ignore Errors:

- Errors can be ignored without handling them in the program. We can do it using pass in except block of error handling section like below.
- -> try:
-      data = something_that_can_go_wrong
- ->except:
-      pass



```
>>> # Ignoring errors
>>> def fun():
...     try:
...         data = 10 / 0
...     except:
...         pass
...     print 'The error was Ignored!'
...
>>> fun()
The error was Ignored!
```

Note: This kind of programming is usually not encouraged. You should know in hand what are the errors that you are going to ignore. Otherwise debugging the errors will be tough.

Built-in Functions:

- Built-in functions are the functions which are built into (already available) Python and can be accessed by end-users.

.

- Some common built-in functions:

» sorted()
» abs()
» all()
» any()
» basestring()
» bin()
» bool()
» enumerate()
» eval()
» file()
» input()
» int()
» len()
» open()
» raw_input()

REGULAR EXPRESSIONS:

- A Regular Expression (abbreviated Regex or Regexp) is a sequence of characters that forms a search pattern, mainly to use in pattern matching with strings, or string matching, i.e: “find and replace” –like operations

- Each character in a regular expression is either a meta-character with its special meaning, or a regular character with its literal meaning.

- RE(Regular Expression) is supported by many programming languages incluing Python

- Python uses built-in “re” module for RegExp work

Why RE?

- RE are portable (almost same syntax in all languages)

.

- They can help you to write shorter code and saves time

- Regular Expressions are fast

- Use regular expressions to:
    - Search a string (Search and Match)
    - Replace part of a string (sub)
    - Break strings into smaller pieces (split)

Why RE?

- RE are portable (almost same syntax in all languages)

.

- They can help you to write shorter code and saves time

- Regular Expressions are fast

- Use regular expressions to:
    - Search a string (Search and Match)
    - Replace part of a string (sub)
    - Break strings into smaller pieces (split)

Basic Syntax:

- Most characters match themselves

  >> The regular expression “test” matches the string ‘test’ , and only that string

- [x] matches any one of a list of characters:

  >> “[abc]” matches ‘a’,’b’,’c’

- [^x] matches any one character that is not included in x

  >> “[^abc]” matches any ingle character except ‘a’,’b’,’c’

  >> “.” matches any single character

- Parentheses can be used for grouping

- >> “(abc)+” matches ‘abc’,’abcabc’,’abcabcabc’ etc

Basic Syntax:

.

- x|y matches x or y
  >> “here|there” matches ‘here’ and ‘there’ , but not ‘herethere’
- x* matches zero or more x’s
  >> “d*” matches ‘ ’,’d’,’dd’,’ddd’,etc
-
- x+ matches one or more x’s
  >> “d*” matches ’d’,’dd’,’ddd’,etc
- x? Matches zero or one x’s
- >> “d?” matches ‘’,or ‘d’
- X{m,n} Matches I x’s , Where m<=i=<n
- >> “d{2,3}” matches ‘dd’,or ‘ddd’

Basic Syntax:

- “\d” matches only digit; “\D” matches any non-digit
- “\s” matches any whitespace character; “\S” matches any non-whitespace chars
- “\w” matches any alphanumeric character; “\W” – non-alphanumeric character
- “^” matches the beginning of the string; “$” matches the end of the string
- “\b” matches a word boundary;“\B” matches position that is not a word boundary

Basic Functions of RE:

- Basic functions of Regular Expressions are:

.

  - re.search()
  - re.match()
  - findall
  - re.sub()
  - re.compile()

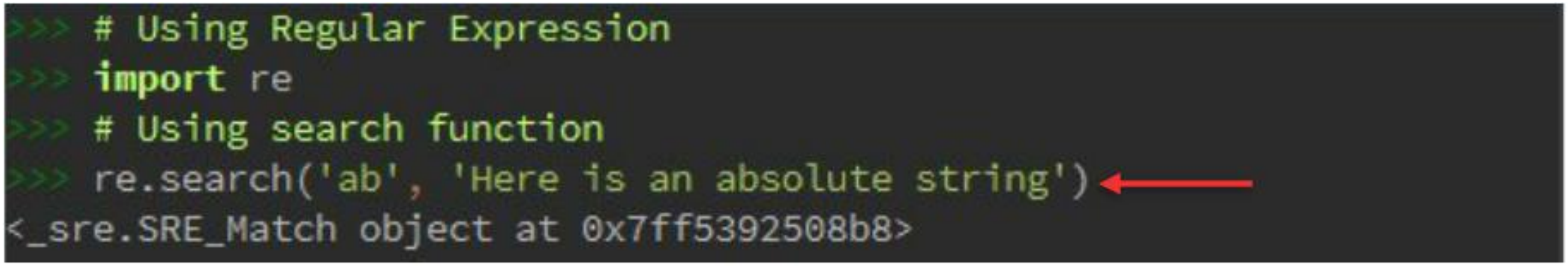
Search and Match:

- Basic functions of Regular Expressions are:

.

>> re.search -> Search looks for a pattern anywhere in a string

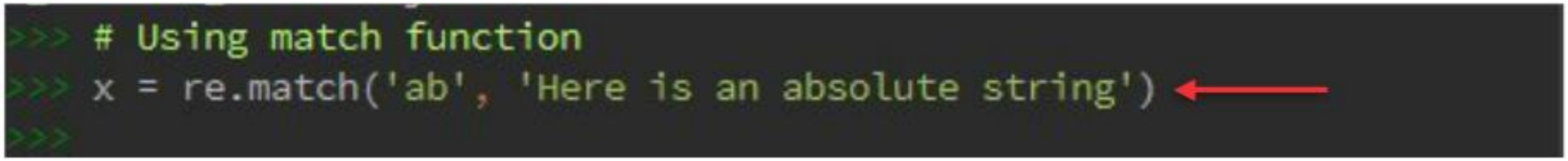
>> syntax: re.search(pattern,string)



```
>>> # Using Regular Expression
>>> import re
>>> # Using search function
>>> re.search('ab', 'Here is an absolute string')
<_sre.SRE_Match object at 0x7ff5392508b8>
```

>> re.match -> Match looks for a match starting at the beginning of the string

>> syntax: re.match(pattern,string)

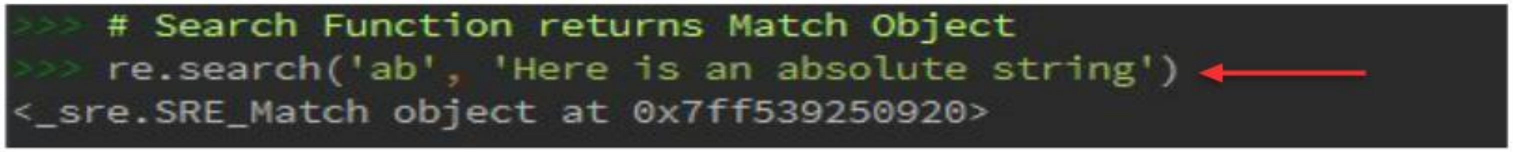


```
>>> # Using match function
>>> x = re.match('ab', 'Here is an absolute string')
>>>
```

The function returns nothing if the match fails, else returns a Match Object.

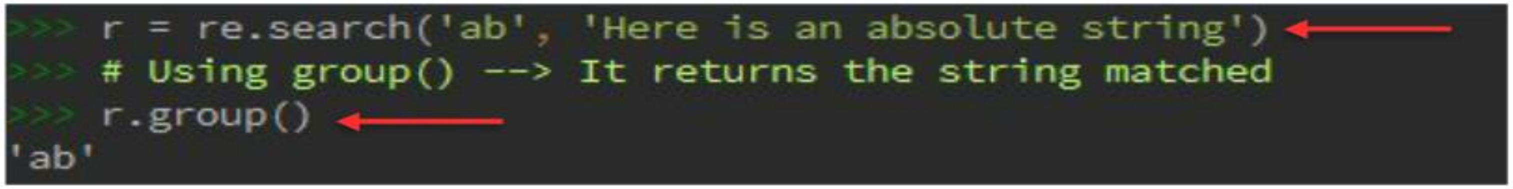
Match Object:

- An Instance of the match class with the details of the match result
- The below operations can be done on a match object



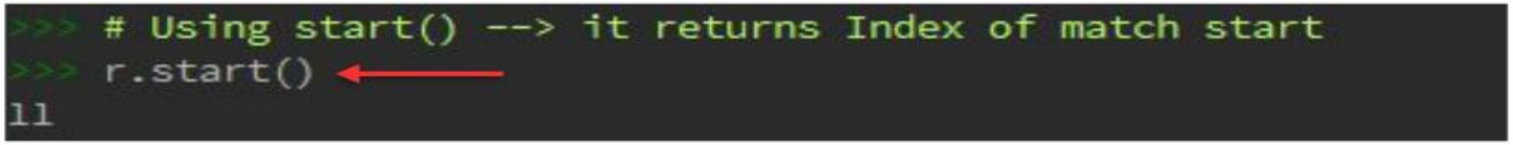
```
>>> # Search Function returns Match Object
>>> re.search('ab', 'Here is an absolute string')
<_sre.SRE_Match object at 0x7ff539250920>
```

Search function returns Match object



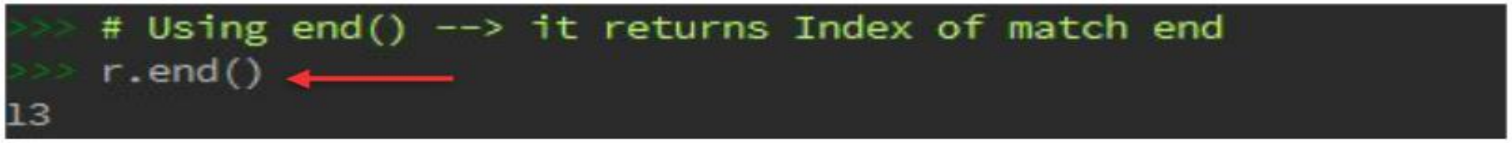
```
>>> r = re.search('ab', 'Here is an absolute string')
>>> # Using group() --> It returns the string matched
>>> r.group()
'ab'
```

Using group() - It returns the string matched



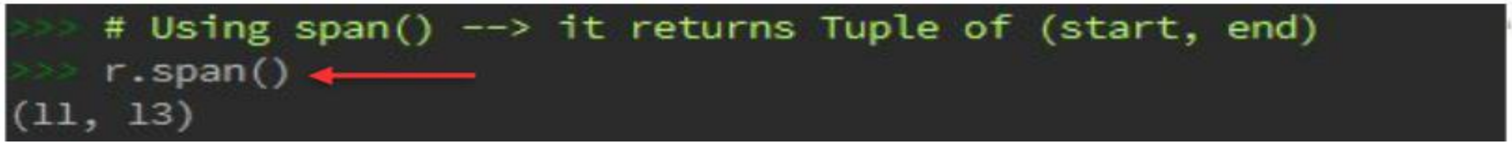
```
>>> # Using start() --> it returns Index of match start
>>> r.start()
11
```

Using start() - It returns the index of match start



```
>>> # Using end() --> it returns Index of match end
>>> r.end()
13
```

Using end() - It returns the index of match end

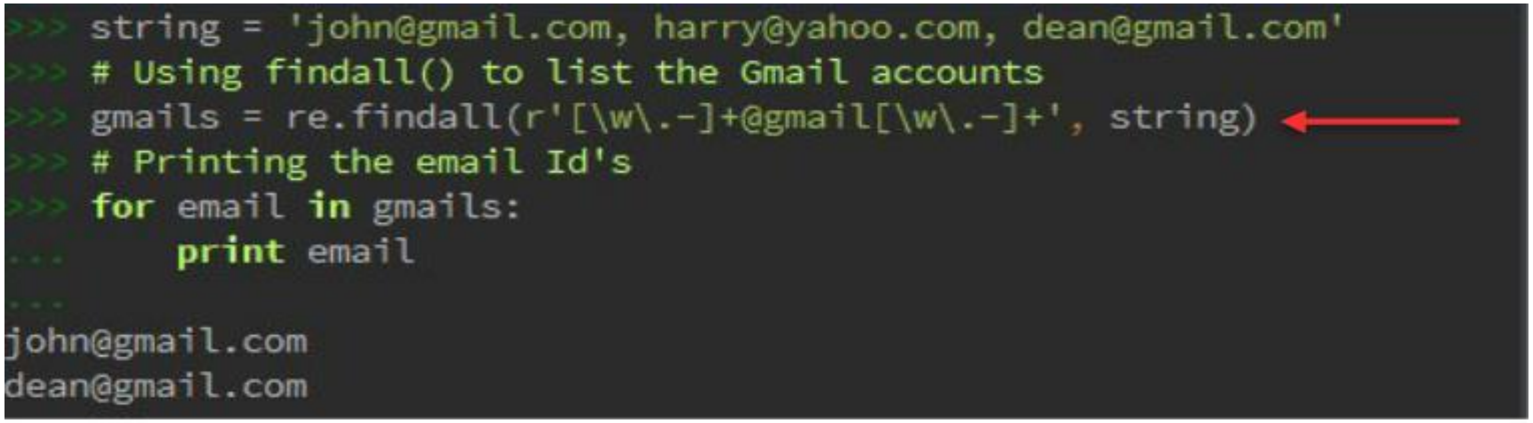


```
>>> # Using span() --> it returns Tuple of (start, end)
>>> r.span()
(11, 13)
```

Using span() - It returns the Tuple of (start, end)

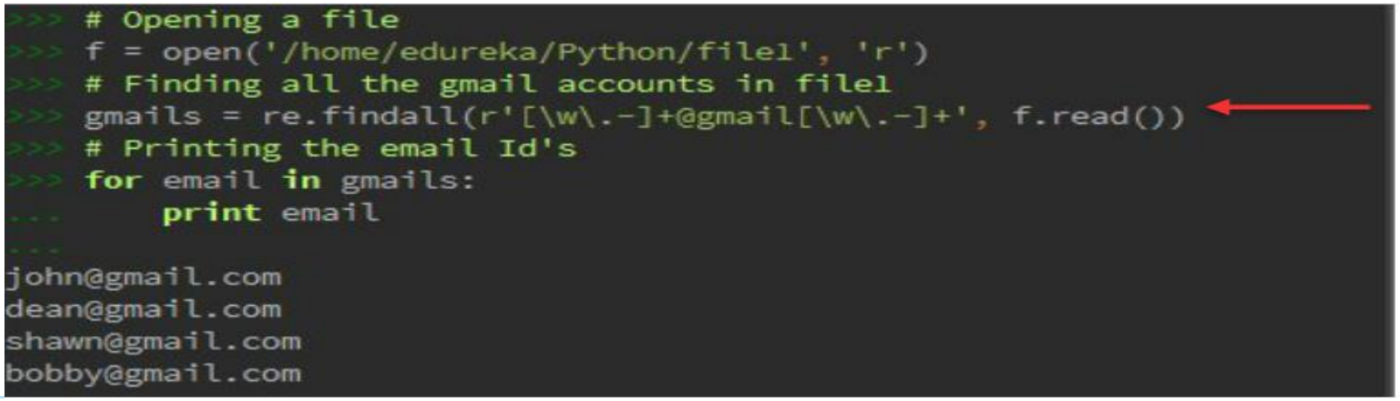
Findall() Function

- Findall() finds *all* the matches and returns them as a list of strings, which each string representing one match



```
>>> string = 'john@gmail.com, harry@yahoo.com, dean@gmail.com'
>>> # Using findall() to list the Gmail accounts
>>> gmails = re.findall(r'[\w\.-]+@gmail[\w\.-]+', string)
>>> # Printing the email Id's
>>> for email in gmails:
...     print email
...
john@gmail.com
dean@gmail.com
```

- Similarly it can be applied on files (No need to iterate over lines to find a match)



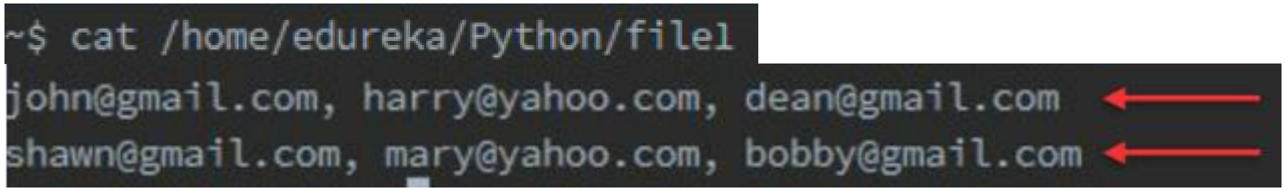
```
>>> # Opening a file
>>> f = open('/home/edureka/Python/file1', 'r')
>>> # Finding all the gmail accounts in file1
>>> gmails = re.findall(r'[\w\.-]+@gmail[\w\.-]+', f.read())
>>> # Printing the email Id's
>>> for email in gmails:
...     print email
...
john@gmail.com
dean@gmail.com
shawn@gmail.com
bobby@gmail.com
```

Findall() Function

- Similarly it can e applied on files (No need to iterate over lines to find a match)

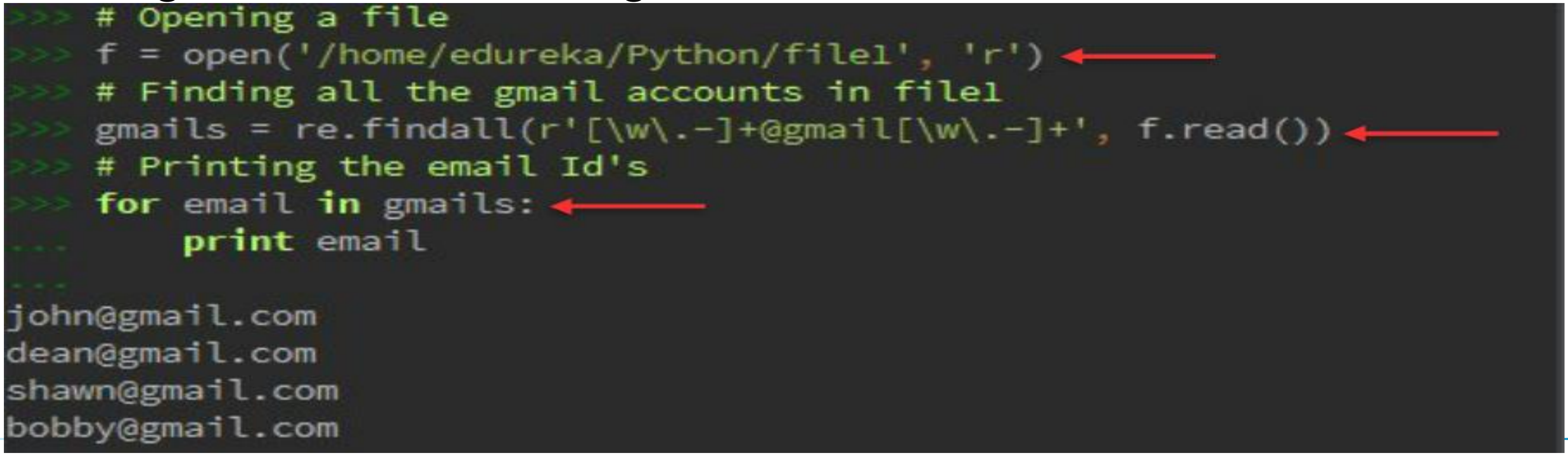
.

>> Content Present in fie1



```
~$ cat /home/edureka/Python/file1
john@gmail.com, harry@yahoo.com, dean@gmail.com
shawn@gmail.com, mary@yahoo.com, bobby@gmail.com
```

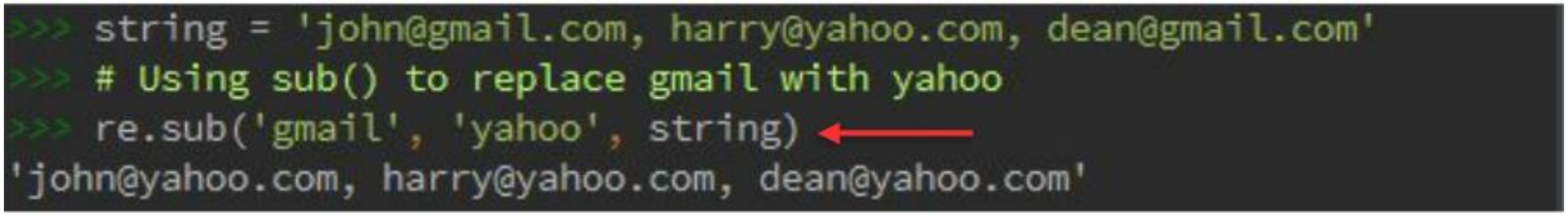
- Reading the file and finding the Gmail account



```
>>> # Opening a file
>>> f = open('/home/edureka/Python/file1', 'r')
>>> # Finding all the gmail accounts in file1
>>> gmails = re.findall(r'[\w\.-]+@gmail[\w\.-]+', f.read())
>>> # Printing the email Id's
>>> for email in gmails:
...     print email
...
john@gmail.com
dean@gmail.com
shawn@gmail.com
bobby@gmail.com
```

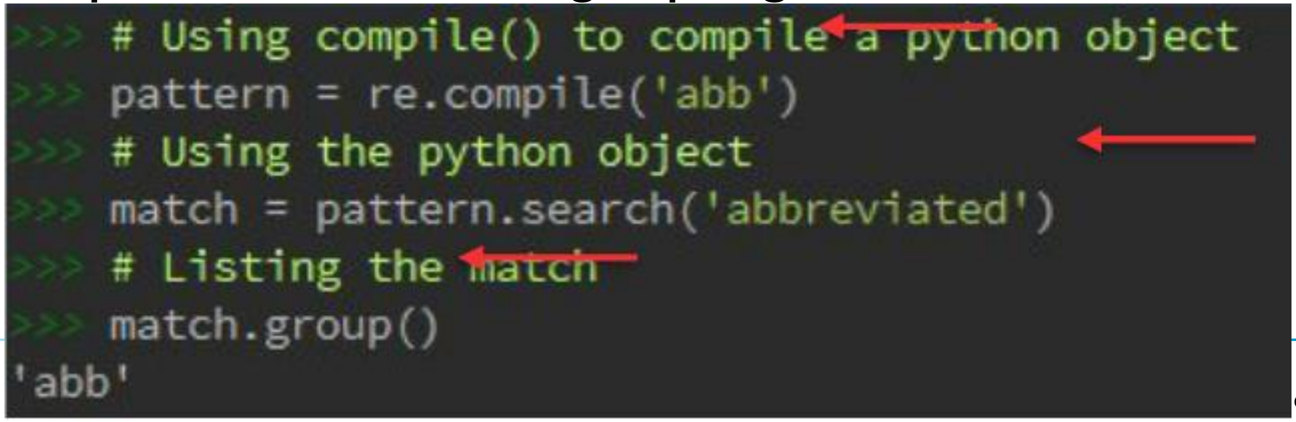
re.sub() and re.compile()

- re.sub() -> Used to replace substrings
- Syntax: re.sub(pattern,repl,string)



```
>>> string = 'john@gmail.com, harry@yahoo.com, dean@gmail.com'
>>> # Using sub() to replace gmail with yahoo
>>> re.sub('gmail', 'yahoo', string)
'john@yahoo.com, harry@yahoo.com, dean@yahoo.com'
```

- re.compile() -> When we compile a regular expression pattern into pattern objects, which can be used for matching.
  - Using a pattern object and reusing it is more efficient when the expression is used multiple times in a single program



```
>>> # Using compile() to compile a python object
>>> pattern = re.compile('abb')
>>> # Using the python object
>>> match = pattern.search('abbreviated')
>>> # Listing the match
>>> match.group()
'abb'
```

Database Access:

▪

SQLite

Relational Database API Specification

- To Access a database, Python needs a database module
- Python database modules that conform to the Python Database API (DB-API) specification can be used to access relational database from Python.
- Python Database API supports a wide range of database server:
  - GadFly
  - SQLite 3
  - mSQL
  - MySQL
  - PostgreSQL
  - Microsott SQL Server 2000
  - Informix
  - Interbase
  - Oracle
  - Sybase

Common Things Across DBs

- Connections

.
>> Every database module provides a module-level function connect(parameter)

>> The Exact parameters depend on the database

>> Example:

conn=connect(dsn=“”hostname:DBNAME”,user=“ramesh”,password=“Passw d12”)

- If successful, a connection object (here conn) is returned
- Following methods can be called using the connection object:

-c.close():

-c.commit():

-c.rollback():

-c.cursor() : Creates a new cursor object that uses the connection, A cursor is an object you can use to execute SQL queries and obtain results

More on Cursor

- An instance cur of a Cursor has a number of standard methods and attributes:

  >>cur.close(): Closes the cursor , Preventing any further operations on it.

  >>cur.execute(query[, parameters]):Executes a query or command query on the database

  >>cur.ftechone():Returns the next row of the result set produced by execute()

  >>cur.fetchmany([size]): Returns a sequence of result rows (e.g./ a list of tuples) . Size is the number of rows to return

  >>cur.fetchall(): Returns a sequence of all remaining result rows(e.g: a list of tuples)

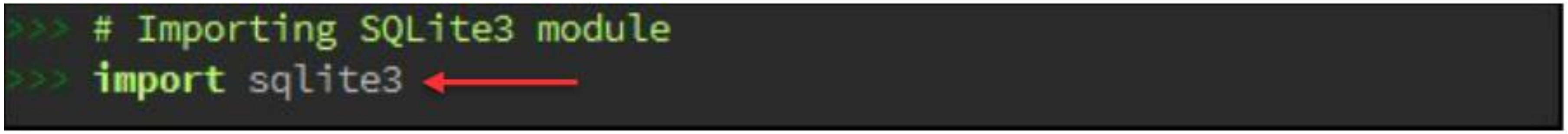
  >>cur.rowcount : The number of rows in the last result produced by one of the execute*() methods.

SQLite DB

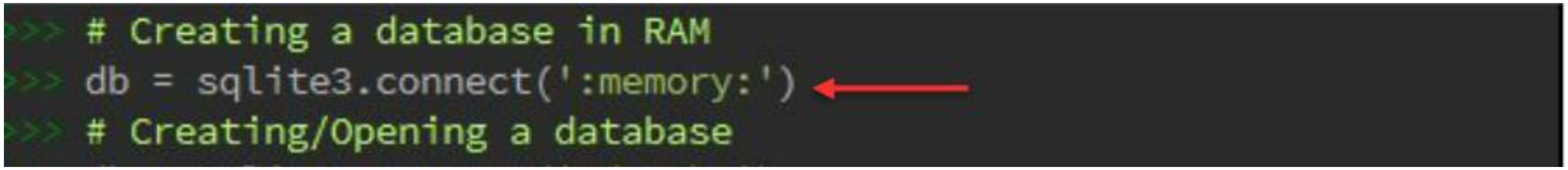
- It is not required to install this module separately as it is being shipped by default along with Python version 2.5.x
- SQLite3 is an easy to use database engine
- It is very fast and lightweight , and the entire database is stored in a single disk file.
- The Python standard library includes a module called “sqlite3”intended to work with this database
- This module is a SQL interface compliant with the DB-API 2.0 specification.

Using SQLite

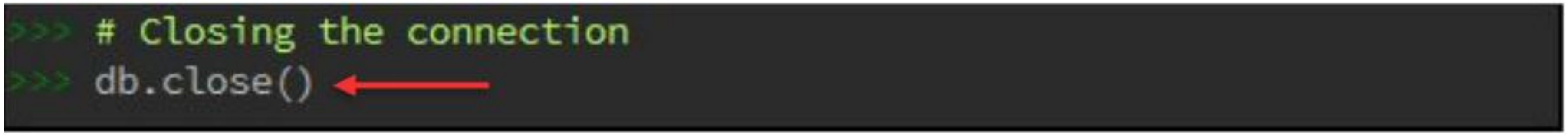
▪



```
>>> # Importing SQLite3 module
>>> import sqlite3
```



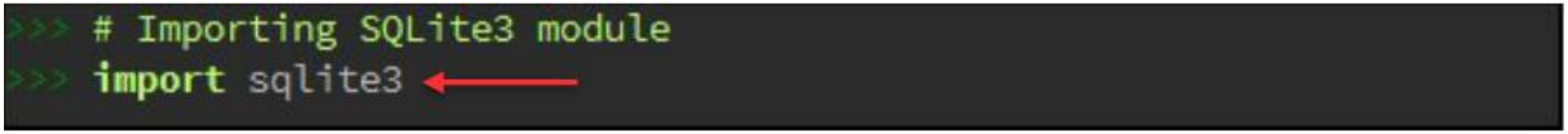
```
>>> # Creating a database in RAM
>>> db = sqlite3.connect(':memory:')
>>> # Creating/Opening a database
```



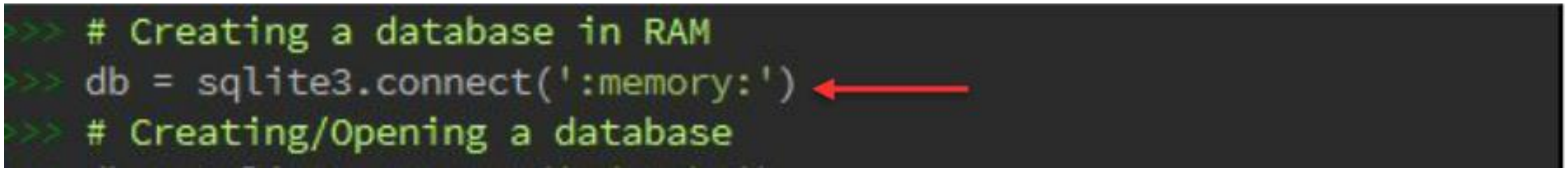
```
>>> # Closing the connection
>>> db.close()
```

Using SQLite

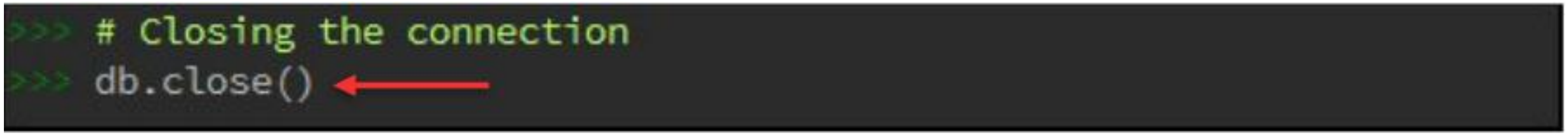
▪



```
>>> # Importing SQLite3 module
>>> import sqlite3
```



```
>>> # Creating a database in RAM
>>> db = sqlite3.connect(':memory:')
>>> # Creating/Opening a database
```



```
>>> # Closing the connection
>>> db.close()
```

Modules:

- A Module is a python file that (generally) has only definitions of variables, functions and classes
.
- Some python modules are written in languages other than python , Most commonly C or C++, Such a module is called an Extension Module
- Importing Module:
    - Import modulename

sys Modules:

- Common Data, Functions present in sys mod
  - >> sys.exit()
  - >> sys.argv

os and subprocess Modules:

- The os and subprocess modules include code that lets python work with your operating system-they even run operating system commands
- The os module in best for the following tasks:
- >> Working with paths and permissions (test for accessing to a path , changing directories, changing access permissions and user/group IDs)
- >> Working with files (open, close, write, truncate, create links)

- The subprocess module, lets you safely interact with operating system to run commands and get information out of them

os Module:

- Commonly used functions in os module:
  - >> os.getcwd()
  - >> os.chdir()
  - >> os.mkdir()
  - >> os.makedirs()
  - >> os.remove()
  - >> os.rmdir()
  - >> os.remvedirs()

Math Number – theoretic Functions:

- math.ceil(x)
- . math.copysign(x,y)
- math.fabs()
- math.exp()
- math.log(x,[, abse])
- math.acos(x)
- math.asin(x)
- math.cos(x)
- math.pi
- math.e

Math Number – theoretic Functions:

- To ease this task various mathematical functions are predefined and can be used using the math module

.

- Math functions are broadly classified into:
- >> Number-theoretic
- >> Power and Logarithmic Functions
- >> Trigonometric Functions
- >> Angular Conversion
- >> Hyperbolic Functions
- >> Special Functions
- >> Constants

Math Number – theoretic Functions:

- To ease this task various mathematical functions are predefined and can be used using the math module

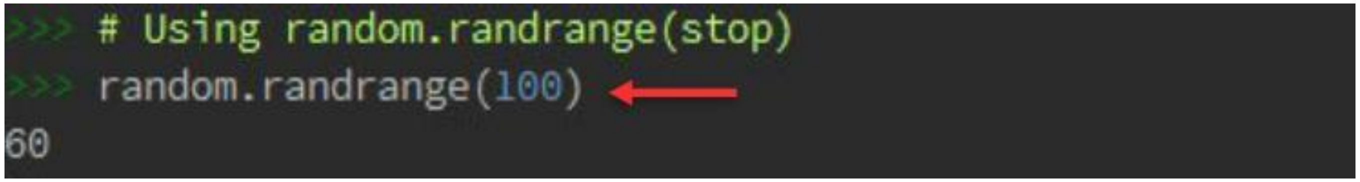
.

- Math functions are broadly classified into:

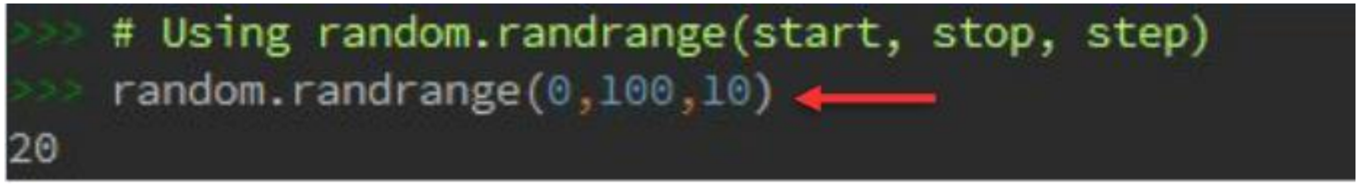
- >> Number-theoretic
- >> Power and Logarithmic Functions
- >> Trigonometric Functions
- >> Angular Conversion
- >> Hyperbolic Functions
- >> Special Functions
- >> Constants

Random:

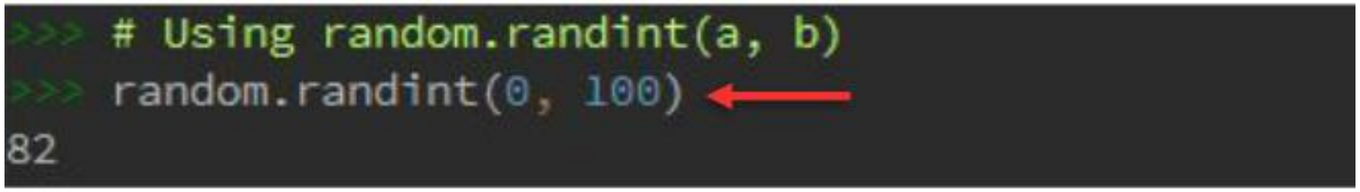
- This module implements pseudo-random number generators for various distributions
- Function for integers:



```
>>> # Using random.randrange(stop)
>>> random.randrange(100)
60
```



```
>>> # Using random.randrange(start, stop, step)
>>> random.randrange(0,100,10)
20
```

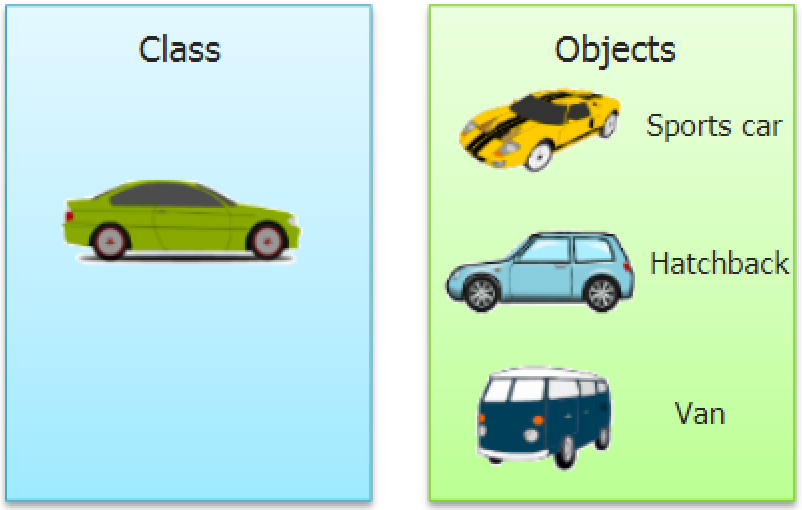


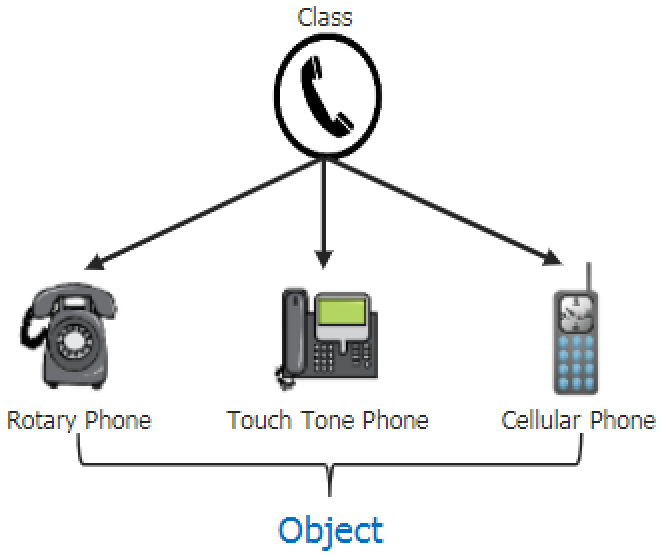
```
>>> # Using random.randint(a, b)
>>> random.randint(0, 100)
82
```

Classes and Objects:

→ Class is a blueprint used to create objects having same property or attribute as its class

→ An Object is an instance of class which contains variables and methods





Classes and Objects:

→Class is a template definition of methods and variables in a particular kind of object

→A Class describes the abstract characteristics of a real-life thing

→There can be instances of Classes

→An instance is an object of a Class created at run-time

→The set of values of the attributes of a particular object is called its state

→A method is a function which belongs to a class

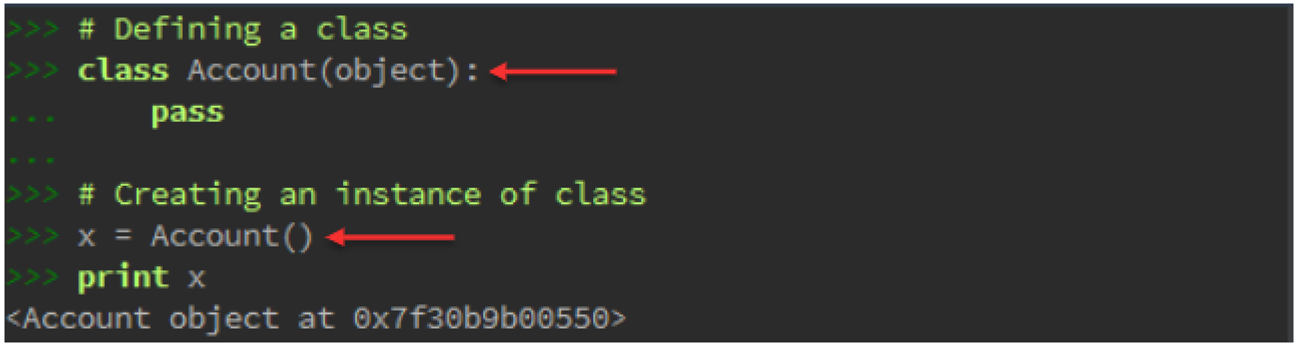
Classes and Objects:

→ All Classes are derived from object

→ Syntax:

```
class Account(object):
        pass
```

→ Now we will create an instance of this empty class:



```
>>> # Defining a class
>>> class Account(object):
...     pass
...
>>> # Creating an instance of class
>>> x = Account()
>>> print x
<Account object at 0x7f30b9b00550>
```

Note: Don't miss the colon after class name

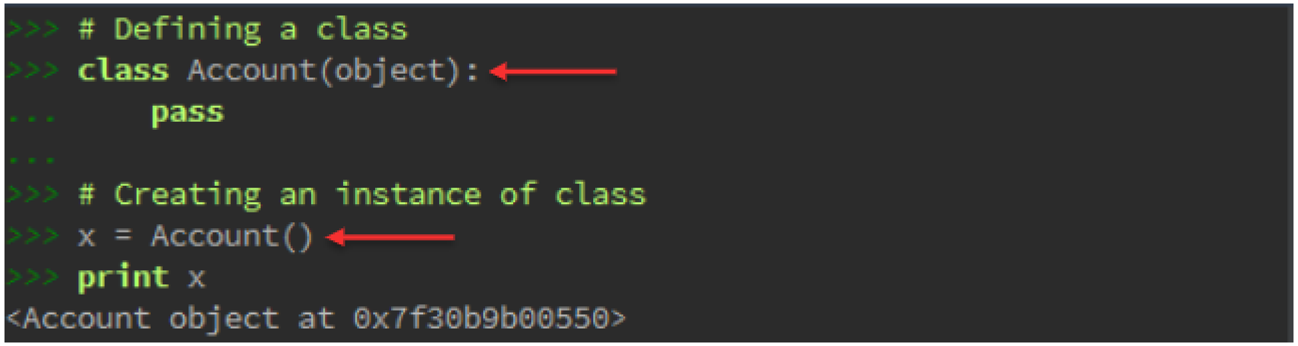
Classes and Objects:

→ All Classes are derived from object

→ Syntax:

```
class Account(object):
        pass
```

→ Now we will create an instance of this empty class:



```
>>> # Defining a class
>>> class Account(object):
...     pass
...
>>> # Creating an instance of class
>>> x = Account()
>>> print x
<Account object at 0x7f30b9b00550>
```

Note: Don't miss the colon after class name

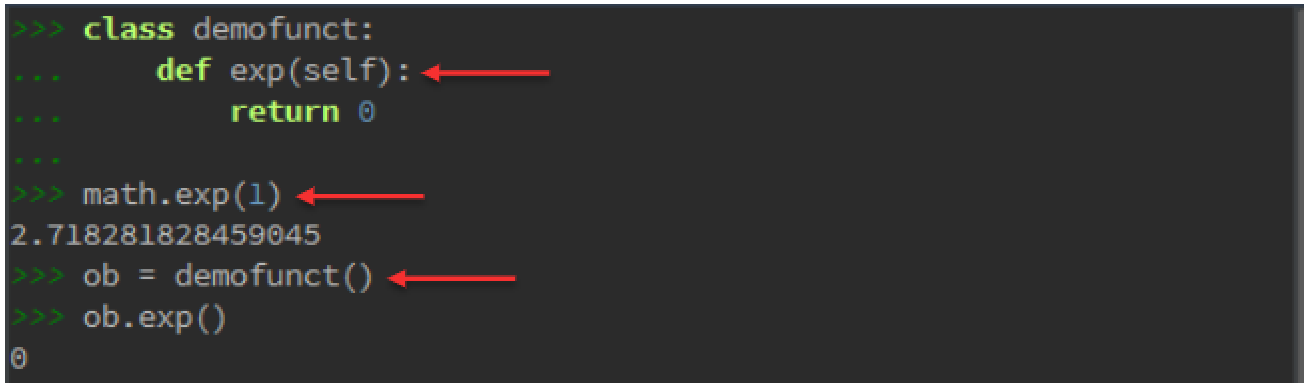
Namespaces

→ A Namespace is a mapping from names to objects

→ A name in Python is analogous to a variable

→ At the simplest level, classes are simply namespaces

→ It can sometimes be useful to put groups of functions in their own namespace to differentiate these functions from other similarly named ones



```
>>> class demofunct:
...     def exp(self):
...         return 0
...
>>> math.exp(1)
2.718281828459045
>>> ob = demofunct()
>>> ob.exp()
0
```

Attributes

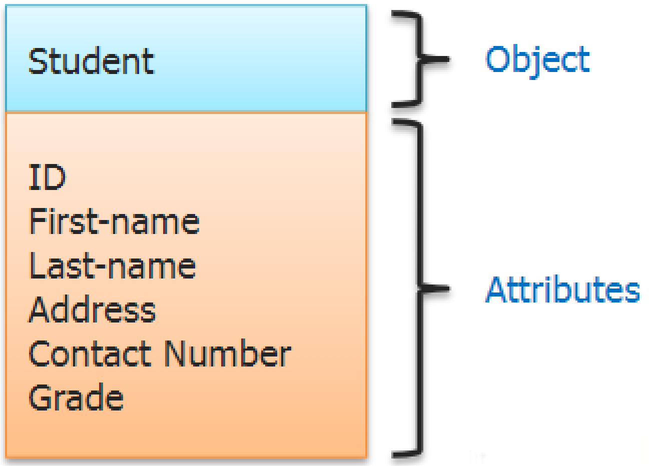
→ "In computing, an attribute is a specification that defines a property of an object, element, or file. It may also refer to or set the specific value for a given instance of such."

- From Wikipedia

→ There are two types of Attributes:

» Built- in Class Attributes

» Attributes defined by Users

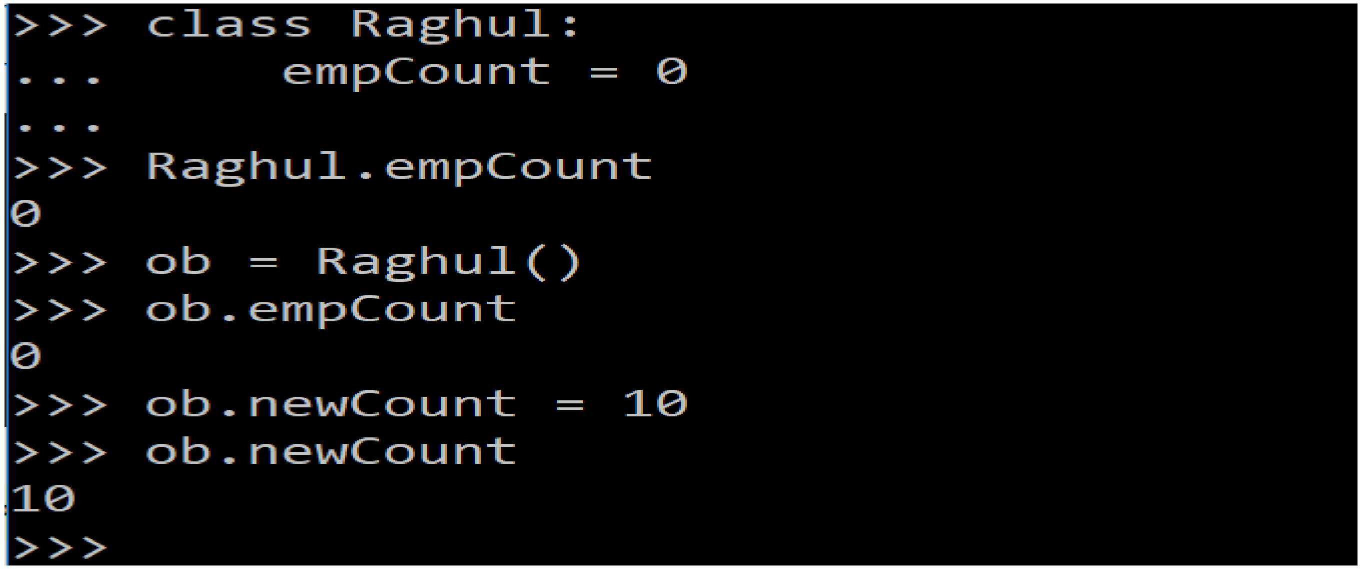


Bulit-in Class Attributes

- __dict__ : Dictionary containing the class's namespace
- __doc__ : Class documentation string or None if undefined
- __name : Class name
- __modue__ : Module name is which the class is defined . This attribute is "__main__" in interactive mode
- __bases__ : A possibly empty tuple containing the base classes , in the order of their occurrences in the base class list

Attribute Defined by Users:

- Attribute defined by users:
  >> Attributes are created inside the class definition
  >> We can dynamically create new attributes of existing instance of a class
  >> Attributes can be bound to class name as well



```
>>> class Raghul:
...     empCount = 0
...
>>> Raghul.empCount
0
>>> ob = Raghul()
>>> ob.empCount
0
>>> ob.newCount = 10
>>> ob.newCount
10
>>>
```

Note: If you try to access a non-existing attribute, you will raise an AttributeError

Public, Protected and Private Attributes

→ Private attributes can only be accessed inside of the class definition

▪

→ Protected (restricted) attributes should only be used under certain conditions

→ Public attributes can and should be freely used

Naming	Type	Meaning
name	Public	These attributes can be freely used inside or outside of a class definition
_name	Protected	Protected attributes should not be used outside of the class definition, unless inside of a subclass definition
__name	Private	This kind of attribute is inaccessible and invisible. It's neither possible to read nor to write those attributes, except inside of the class definition itself

Public, Protected and Private Attributes

→ Private attributes can only be accessed inside of the class definition

▪

→ Protected (restricted) attributes should only be used under certain conditions

→ Public attributes can and should be freely used

Naming	Type	Meaning
name	Public	These attributes can be freely used inside or outside of a class definition
_name	Protected	Protected attributes should not be used outside of the class definition, unless inside of a subclass definition
__name	Private	This kind of attribute is inaccessible and invisible. It's neither possible to read nor to write those attributes, except inside of the class definition itself

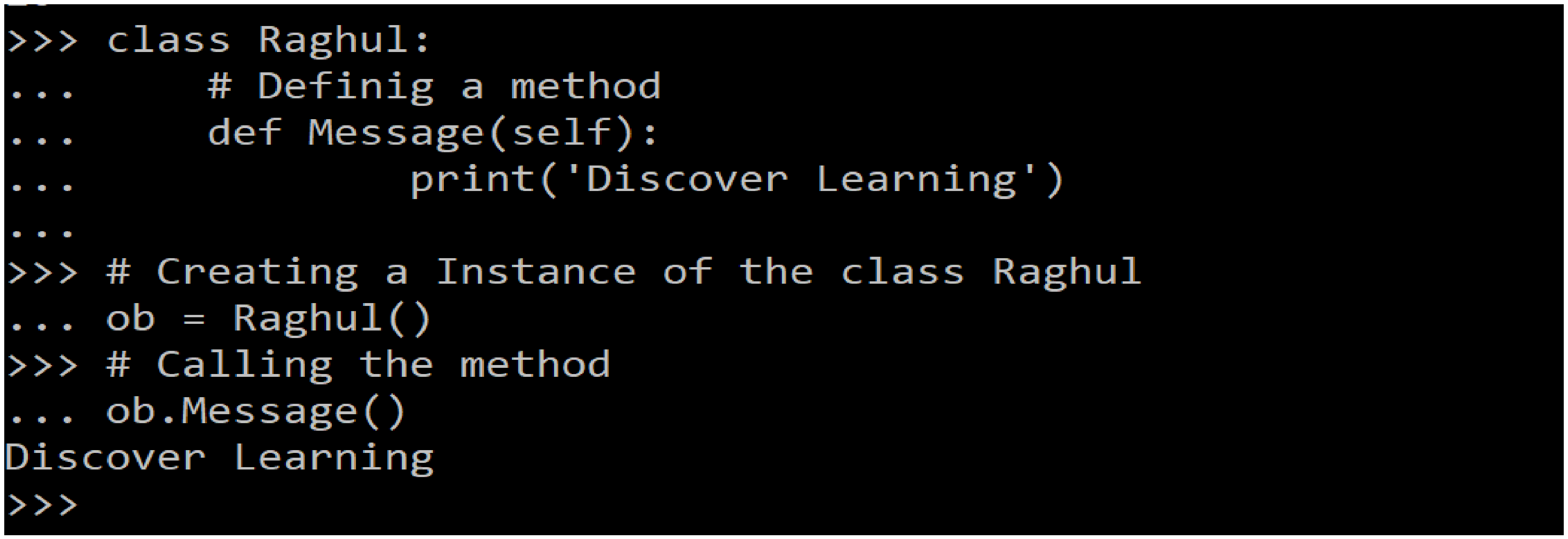
Class Variable & Instance Variables

- Class variable are shared by all Instances
- Instance variable are unique to each instance

Definition of Methods

→ A Method differs from a function only in two aspects:

- » It is defined within a class
- » The first parameter in the definition of a method has to be a reference "self" to the instance of the class
- » A method is called without "self" parameter



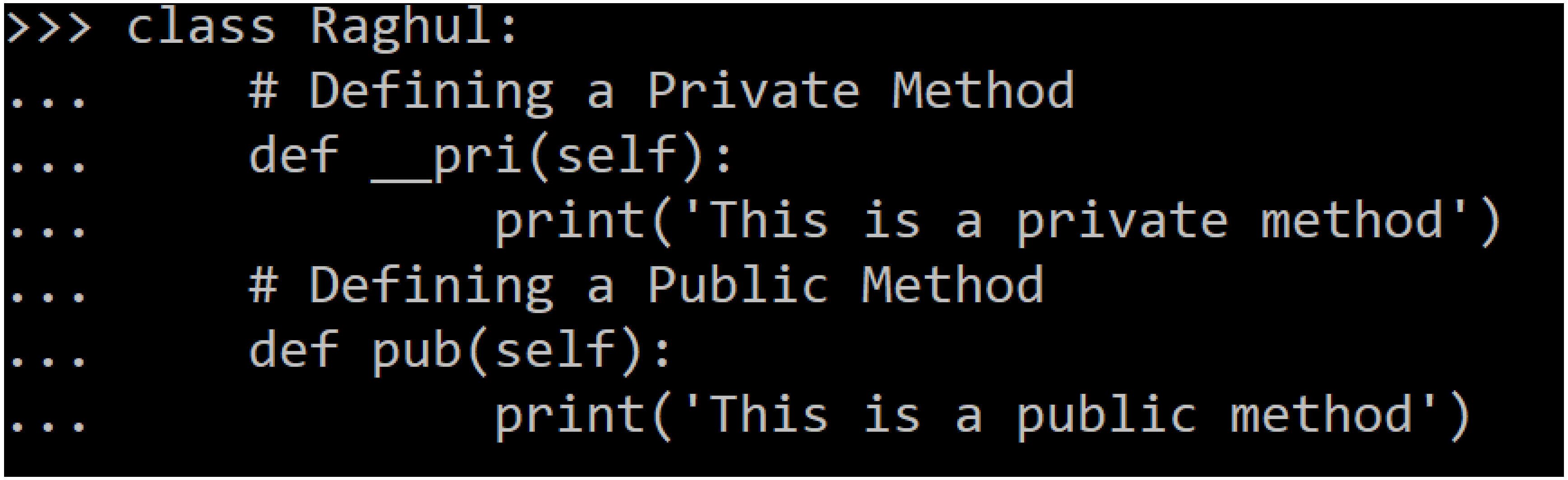
```
>>> class Raghul:
...     # Definig a method
...     def Message(self):
...             print('Discover Learning')
...
>>> # Creating a Instance of the class Raghul
... ob = Raghul()
>>> # Calling the method
... ob.Message()
Discover Learning
>>>
```

Private Methods in Python:

→ When the attributes of an object can only be accessed inside the class, it is called Private

→ Python use two underscores to hide a Method

→ Two underscores can also be used to hide a Variable

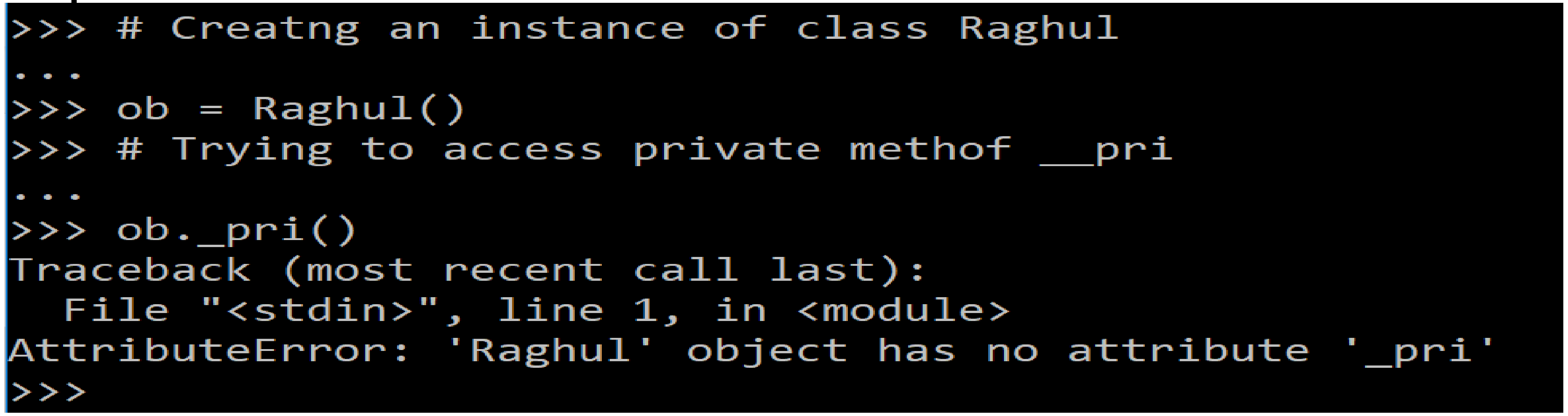


```
>>> class Raghul:
...     # Defining a Private Method
...     def __pri(self):
...             print('This is a private method')
...     # Defining a Public Method
...     def pub(self):
...             print('This is a public method')
...
```

Private Methods Cont…

- If we try to access the Private method, we will get an error

Example:



```
>>> # Creatng an instance of class Raghul
...
>>> ob = Raghul()
>>> # Trying to access private methof __pri
...
>>> ob._pri()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Raghul' object has no attribute '_pri'
>>>
```

Static Method

→ Simple functions with no 'self' argument

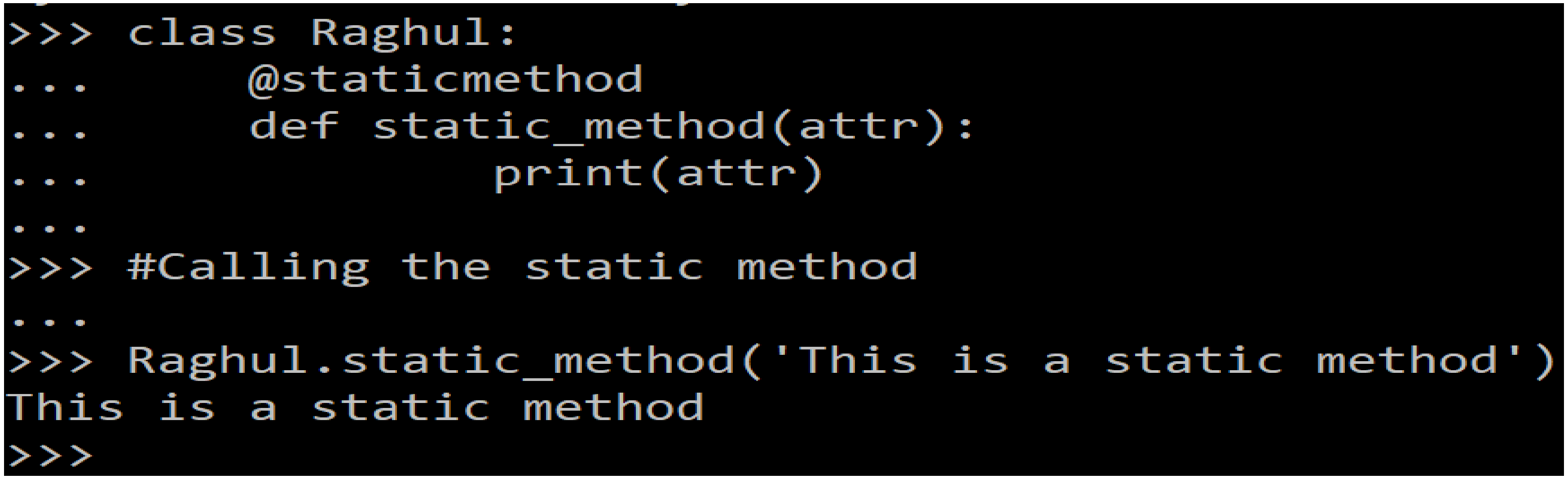
→ Nested inside a class

→ It is a function that has nothing to do with the instance but still belongs to the class

→ Works on Class attributes and not on Instance attribute

→ Can be called through both Class and Instance

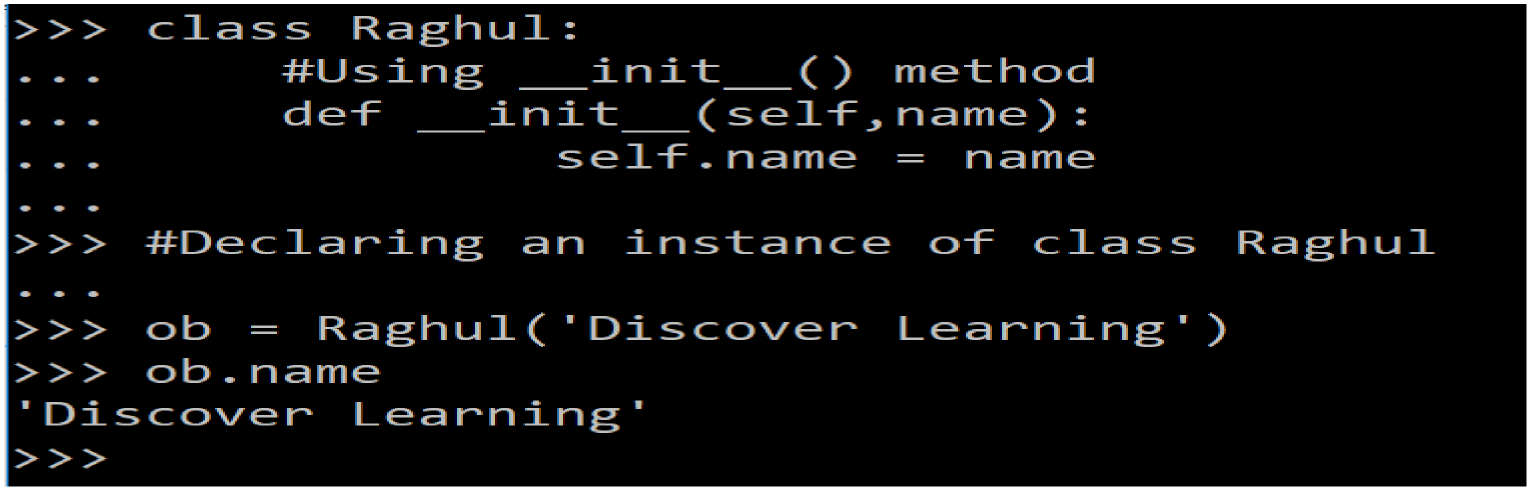
→ Decorator @staticmethod is used to create a static method



```
>>> class Raghul:
...       @staticmethod
...       def static_method(attr):
...               print(attr)
...
>>> #Calling the static method
...
>>> Raghul.static_method('This is a static method')
This is a static method
>>>
```

Constructor – The __init__ Method

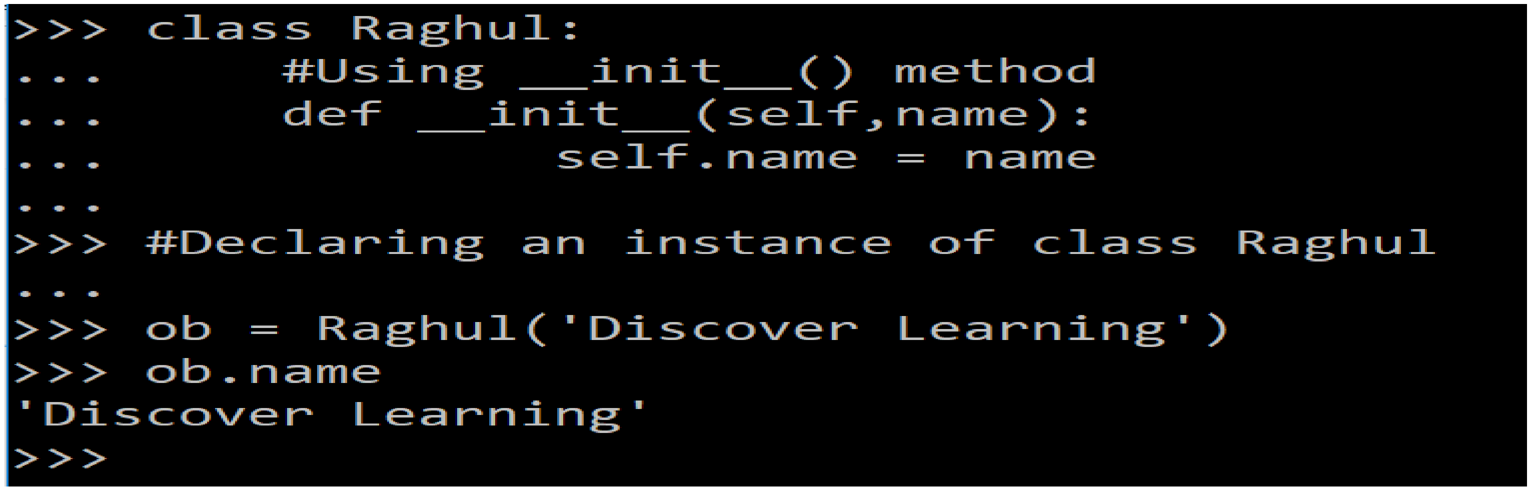
- Define the attribute of an instance right after its creation
- Is immediately and automatically called after an Instance has been created
- It is usually the first method of a class , i.e: it follows right after the class header
- Sometimes called as Constructor



```
>>> class Raghul:
...     #Using __init__() method
...     def __init__(self,name):
...             self.name = name
...
>>> #Declaring an instance of class Raghul
...
>>> ob = Raghul('Discover Learning')
>>> ob.name
'Discover Learning'
>>>
```

Constructor – The __init__ Method

- Define the attribute of an instance right after its creation
- Is immediately and automatically called after an Instance has been created
- It is usually the first method of a class , i.e: it follows right after the class header
- Sometimes called as Constructor



```
>>> class Raghul:
...     #Using __init__() method
...     def __init__(self,name):
...             self.name = name
...
>>> #Declaring an instance of class Raghul
...
>>> ob = Raghul('Discover Learning')
>>> ob.name
'Discover Learning'
>>>
```

Object Oriented Programming

→ Object Oriented Design focuses on:

   » Encapsulation

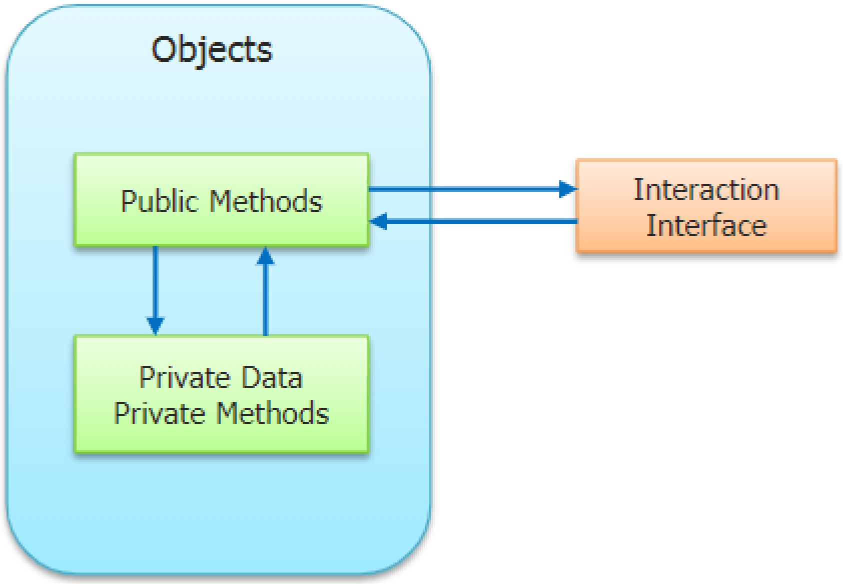
   » Inheritance

   » Polymorphism

→ We will discuss each one of these in detail in further slides

Encapsulation

→ Encapsulation: Dividing the code into a public interface, and a private implementation of that interface

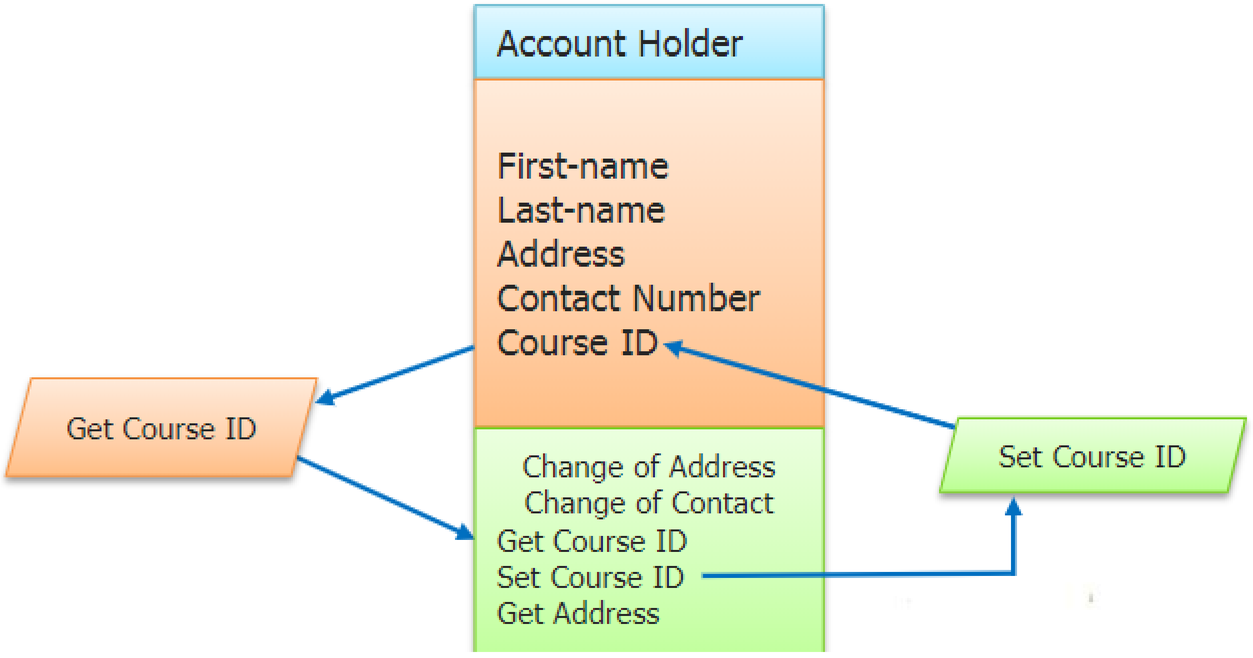


Encapsulation (Contd.)

→ Encapsulation is the mechanism for restricting the access to some of an object's components

→ Access to this data is typically only achieved through special methods: Getters and Setters

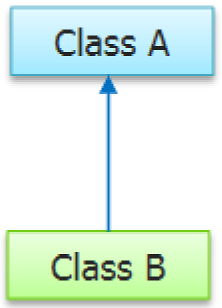
→ By using solely get() and set() methods, we can make sure that the internal data cannot be accidentally set into an inconsistent or invalid state



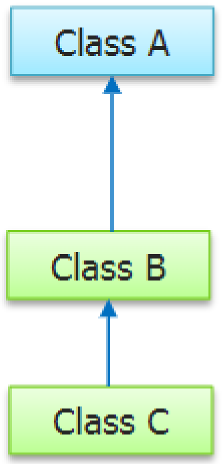
Inheritance

→ Inheritance: The ability to create Sub-classes that contain specializations of their parents

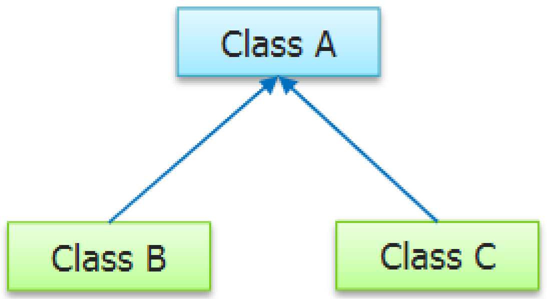
→ Types of Inheritance:



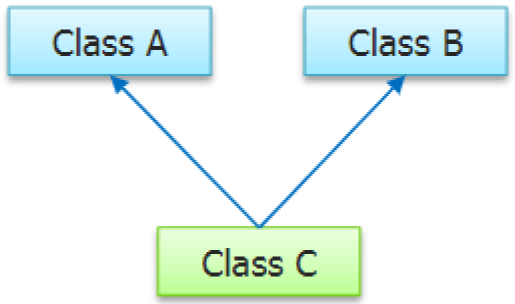
Single Inheritance



Multilevel Inheritance



Hierarchical Inheritance



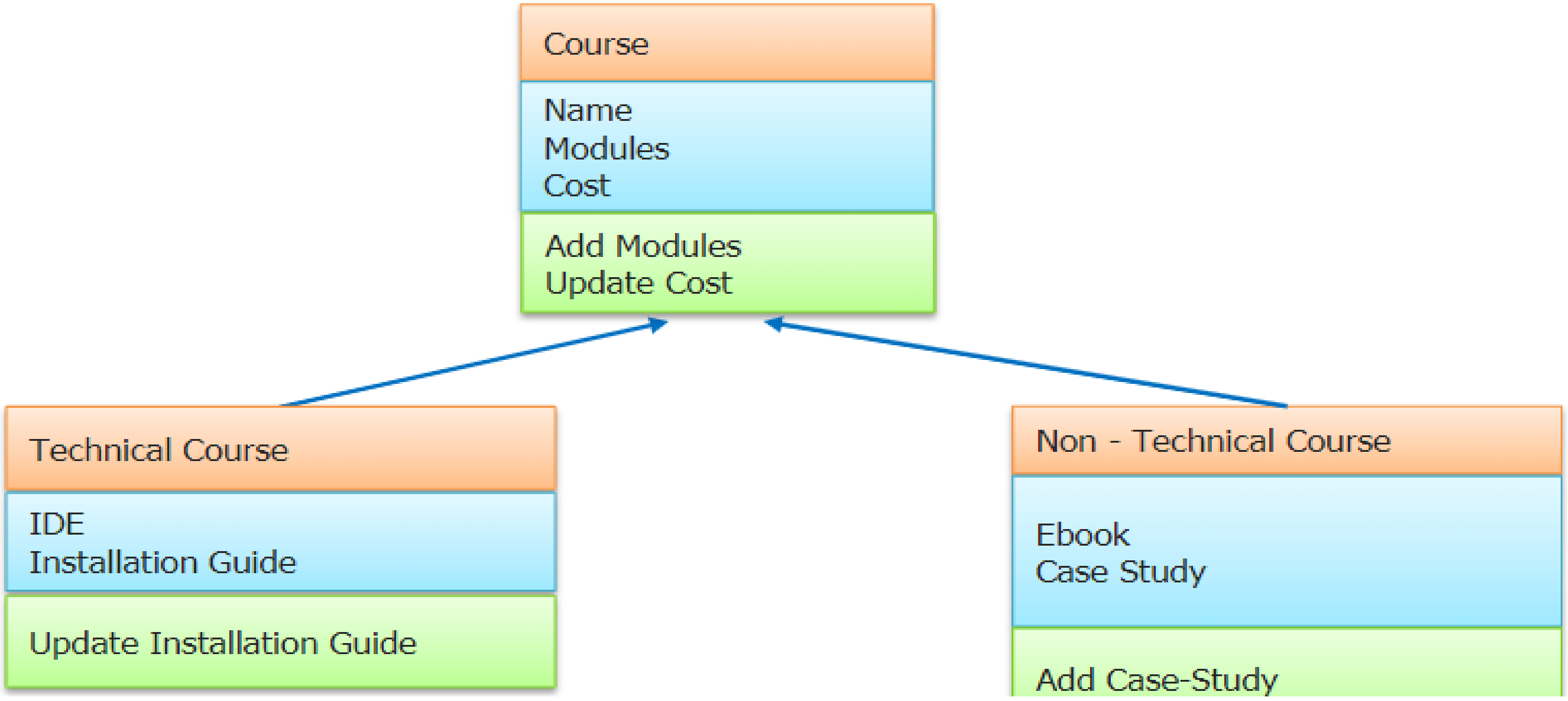
Multiple Inheritance

Inheritance(Contd.)

→ Classes can inherit other classes

→ A Class can inherit attributes and behaviour (methods) from other classes, called Super-classes

→ A Class which inherits from Super-classes is called a Sub-class



Inheritance in Python :

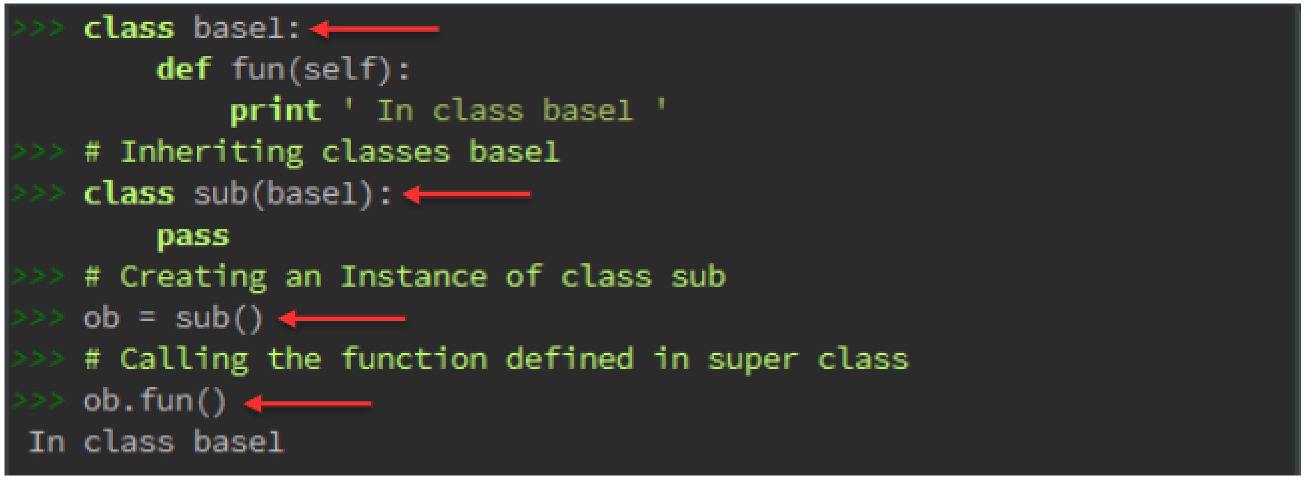
→Important benefits of inheritance are code reuse and reduction of complexity of a program

→Syntax for a subclass definition looks like this:

Syntax:

```
class DerivedClassName(BaseClassName1, BaseClassName2,....):
    pass
```

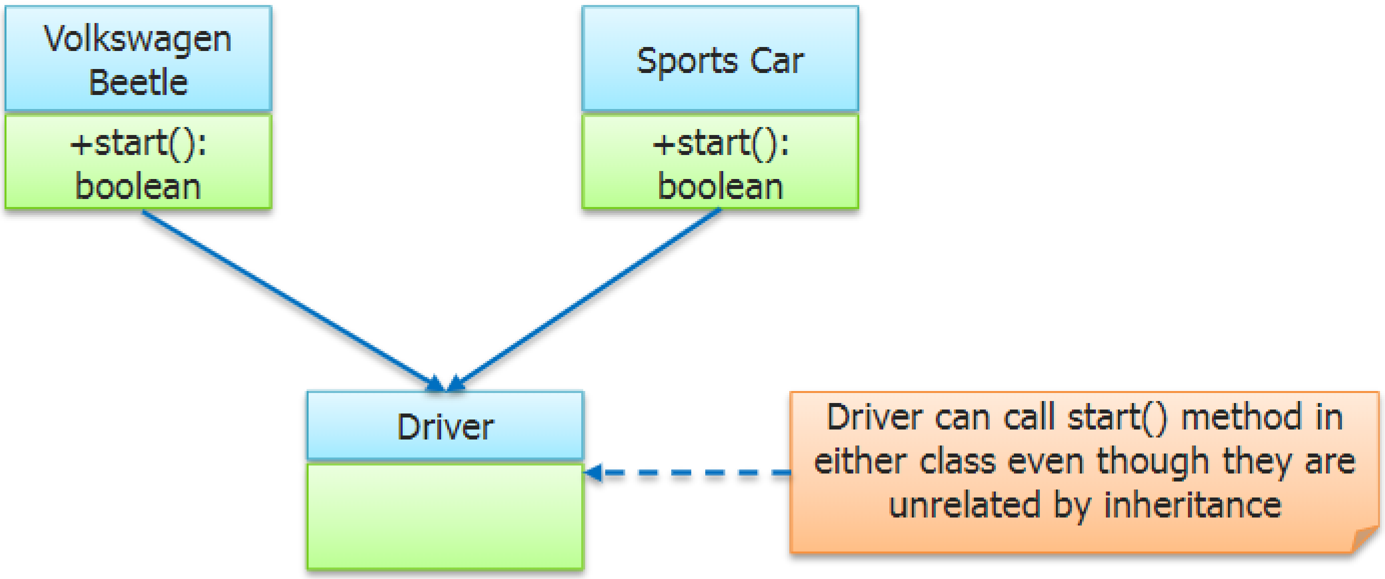
Example:



```
>>> class base1:
        def fun(self):
            print ' In class base1 '
>>> # Inheriting classes base1
>>> class sub(base1):
        pass
>>> # Creating an Instance of class sub
>>> ob = sub()
>>> # Calling the function defined in super class
>>> ob.fun()
 In class base1
```

Polymorphism

→ Polymorphism: The ability to overload standard operators so that they have appropriate behavior based on their context


Volkswagen Beetle
+start(): boolean
Sports Car
+start(): boolean
Driver
Driver can call start() method in either class even though they are unrelated by inheritance

Polymorphism

→ Polymorphism in Computer Science is the ability to present the same interface for differing underlying forms

→ In practical terms, polymorphism means that if class B inherits from class A, it doesn't have to inherit everything about class A; it can do some of the things that class A does differently

→ Polymorphism is most commonly used when dealing with inheritance

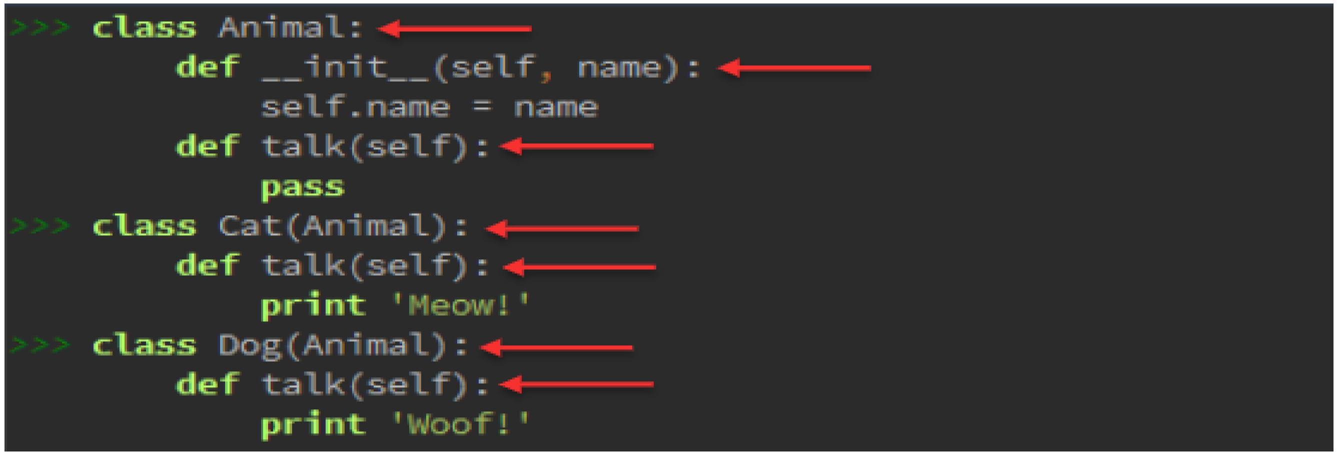
→ Python is implicitly polymorphic

Example:

Here we use the same indexing operator for three different data types

```
a = "alfa"
b = (1, 2, 3, 4)
c = ['o', 'm', 'e', 'g', 'a']

print a[2]
print b[1]
print c[3]
```

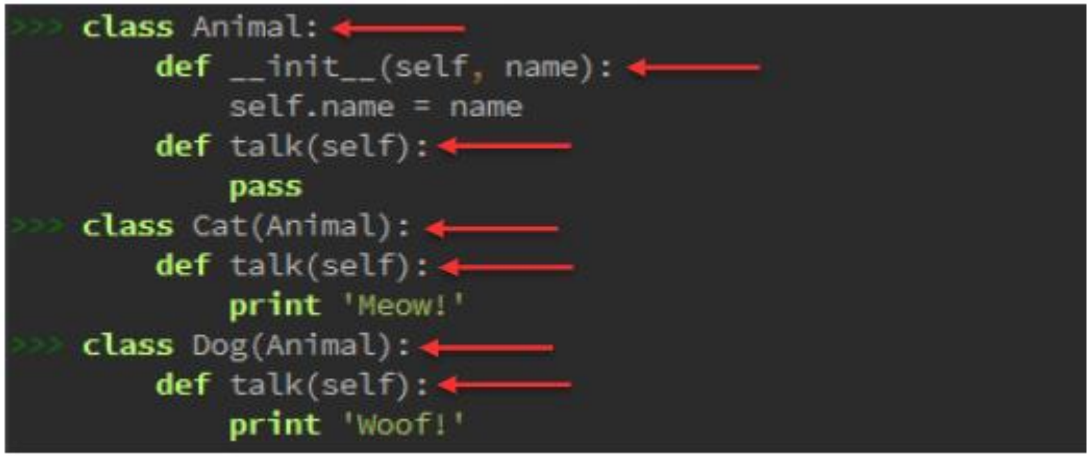
Polymorphism (Contd.)



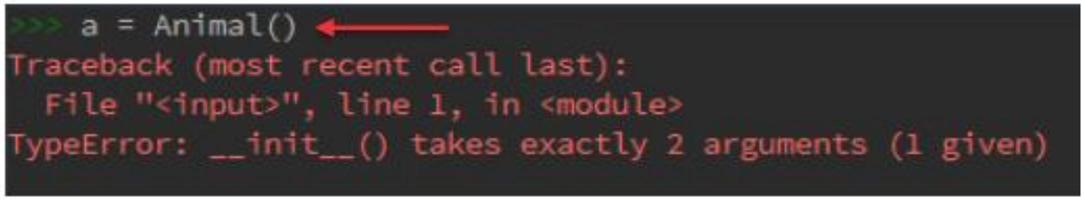
```
>>> class Animal:
        def __init__(self, name):
            self.name = name
        def talk(self):
            pass
>>> class Cat(Animal):
        def talk(self):
            print 'Meow!'
>>> class Dog(Animal):
        def talk(self):
            print 'Woof!'
```

```
a = Animal()
a.talk()      # Guess the o/p  ??

c = Cat("Kitty")
c.talk()      # Guess the o/p  ??

d = Dog("Tommy")
d.talk()    # Guess the o/p  ??
```

Polymorphism (Contd.)



```
>>> class Animal:
        def __init__(self, name):
            self.name = name
        def talk(self):
            pass
>>> class Cat(Animal):
        def talk(self):
            print 'Meow!'
>>> class Dog(Animal):
        def talk(self):
            print 'Woof!'
```



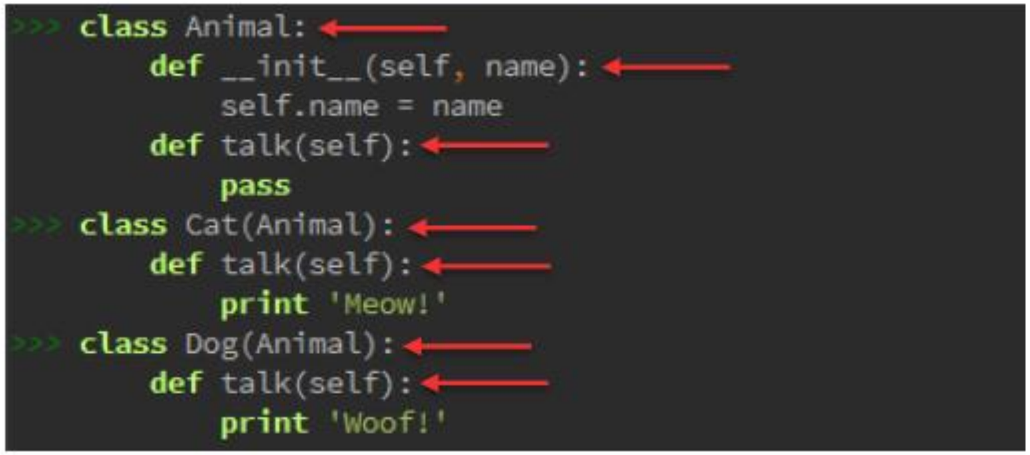
```
>>> a = Animal()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: __init__() takes exactly 2 arguments (1 given)
```

a = Animal()
a.talk()   # Guess the o/p ??

c = Cat("Kitty")
c.talk()   # Guess the o/p ??

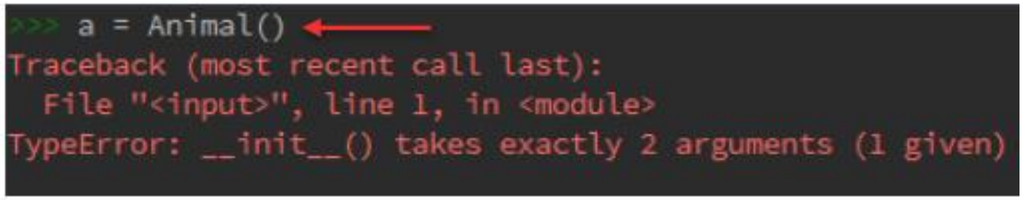
d = Dog("Tommy")
d.talk()   # Guess the o/p ??

Polymorphism (Contd.)



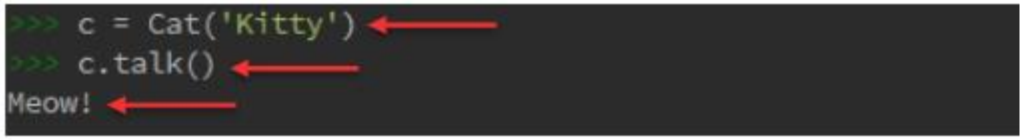
```
>>> class Animal:
        def __init__(self, name):
            self.name = name
        def talk(self):
            pass
>>> class Cat(Animal):
        def talk(self):
            print 'Meow!'
>>> class Dog(Animal):
        def talk(self):
            print 'Woof!'
```

```
a = Animal()
a.talk()      # Guess the o/p  ??
```



```
>>> a = Animal()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: __init__() takes exactly 2 arguments (1 given)
```

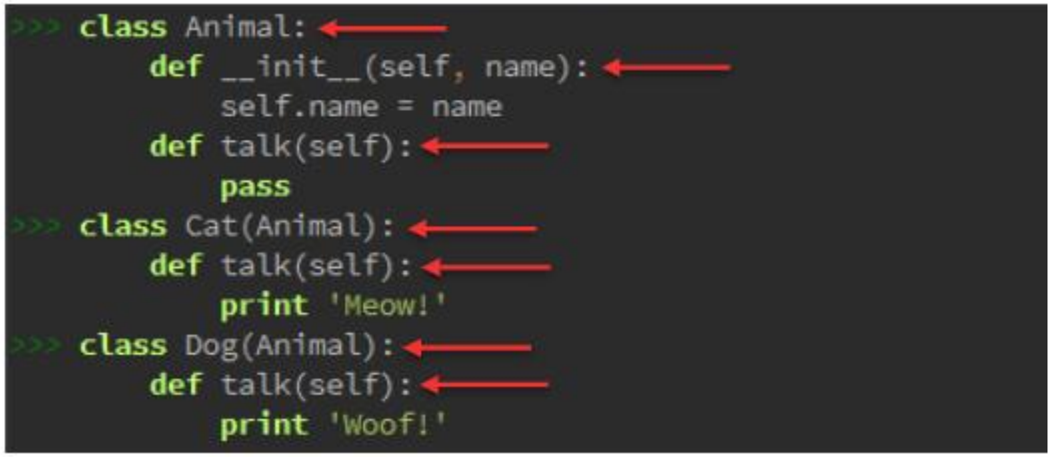
```
c = Cat("Kitty")
c.talk()      # Guess the o/p  ??
```



```
>>> c = Cat('Kitty')
>>> c.talk()
Meow!
```

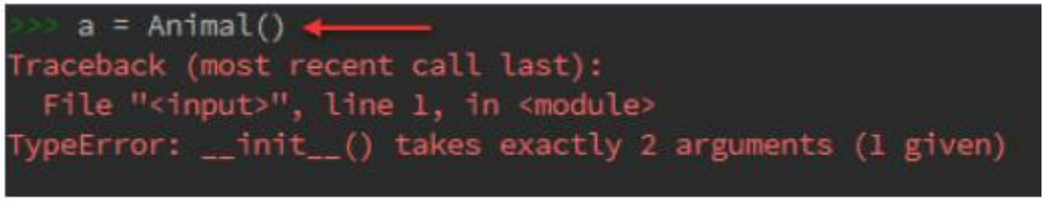
```
d = Dog("Tommy")
d.talk()    # Guess the o/p  ??
```

Polymorphism (Contd.)



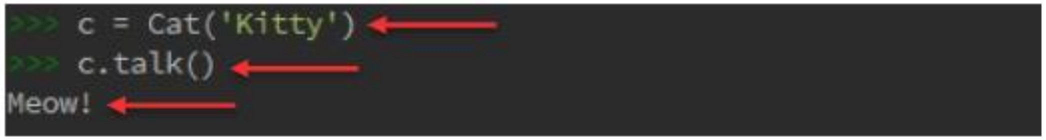
```
>>> class Animal:
        def __init__(self, name):
            self.name = name
        def talk(self):
            pass
>>> class Cat(Animal):
        def talk(self):
            print 'Meow!'
>>> class Dog(Animal):
        def talk(self):
            print 'Woof!'
```

```
a = Animal()
a.talk()      # Guess the o/p  ??
```



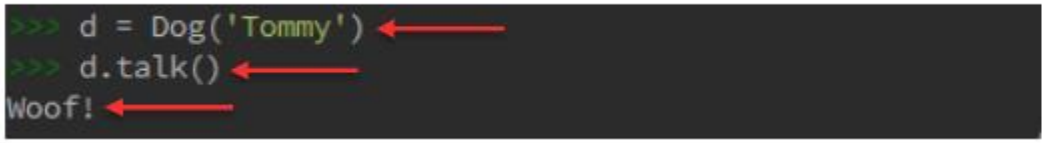
```
>>> a = Animal()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: __init__() takes exactly 2 arguments (1 given)
```

```
c = Cat("Kitty")
c.talk()      # Guess the o/p  ??
```



```
>>> c = Cat('Kitty')
>>> c.talk()
Meow!
```

```
d = Dog("Tommy")
d.talk()    # Guess the o/p  ??
```



```
>>> d = Dog('Tommy')
>>> d.talk()
Woof!
```

Special Methods:

→ These methods are not called directly, but by a specific language syntax

→ This is similar to what is known as operator overloading in C++ or Ruby

→ __init__(), __str__(), __len__() and the __del__() methods in the below example

Questions ?