# Programming Python 3

Giridhar Sripathi

# Course Overview

## Introduction to Python Programming

### Course Scope

- Introduce to Python version 3 programming basics
- Present tools and technics necessary to develop scripts in Python language
- Present major Python development areas today
- Indicate future advanced programming skills

# Course Overview

- Declare variables
- Construct arithmetical and logical expressions
- Use lists and dictionaries
- Use conditional statements
- Manipulate text using regular expression
- Declare and use modules
- Use and create Python Objects
- Use built-in library functions
- Read/write text and binary files
- Interact with the email system
- Interact with Databases

# A Brief History of Python

- ABC is a general-purpose programming language and programming environment, which had been developed in the Netherlands, Amsterdam, at the CWI (Centrum Wiskunde & Informatica)

- The greatest achievement of ABC was to influence the design of Python

- Python was conceptualised in the late 1980s

# A Brief History of Python

- **February 1991**
  First version of Python code (version 0.9.0)

- **January 1994**
  Python version 1.0

- **October 2000**
  Python 2.0

- **2008**
  Python 3.0. Also known as Python 3000 and Py3K

# Why Use Python?

- ✓ **Python Is Easy to Use**
  - So straightforward that it has been called “programming at the speed of thought”
  - Only 33 key-words used in Python language

- ✓ **Python Is Powerful**
  - Simple and Efficient for most of programming tasks
  - Adopted by both programmers and non-programmers

- ✓ **Python is Object-Oriented**
  - *Object-oriented programming* (OOP) is a modern approach to solving problems with computers

# Why Use Python?

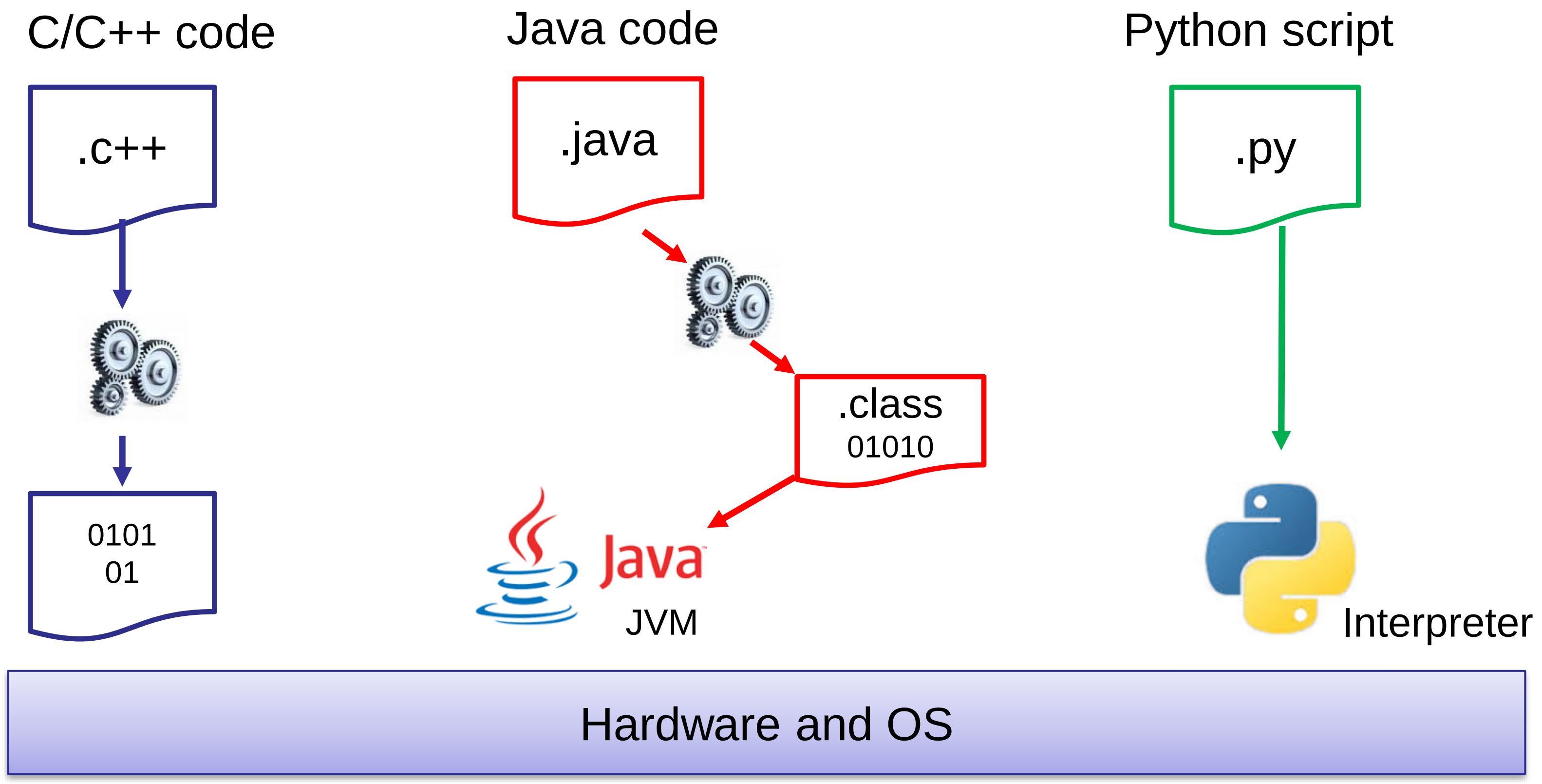
- ✓ **Python is Open Source**
  - OOS **source** code is made available to anyone to study, change and distribute

- ✓ **Python is a “Glue” Language**
  - Can be integrated with other languages i.e. C, C++ and Java
  - A programmer can take advantage of work already done in another language while using Python

- ✓ **Python is Platform Independent**
  - Python runs everywhere, from Windows PC, Linux, Android, etc
  - Large set of modules in standard library and community

# Python Platform

## How Python runs vs C/C++ vs Java



## Python Flavours

There are few types of Python

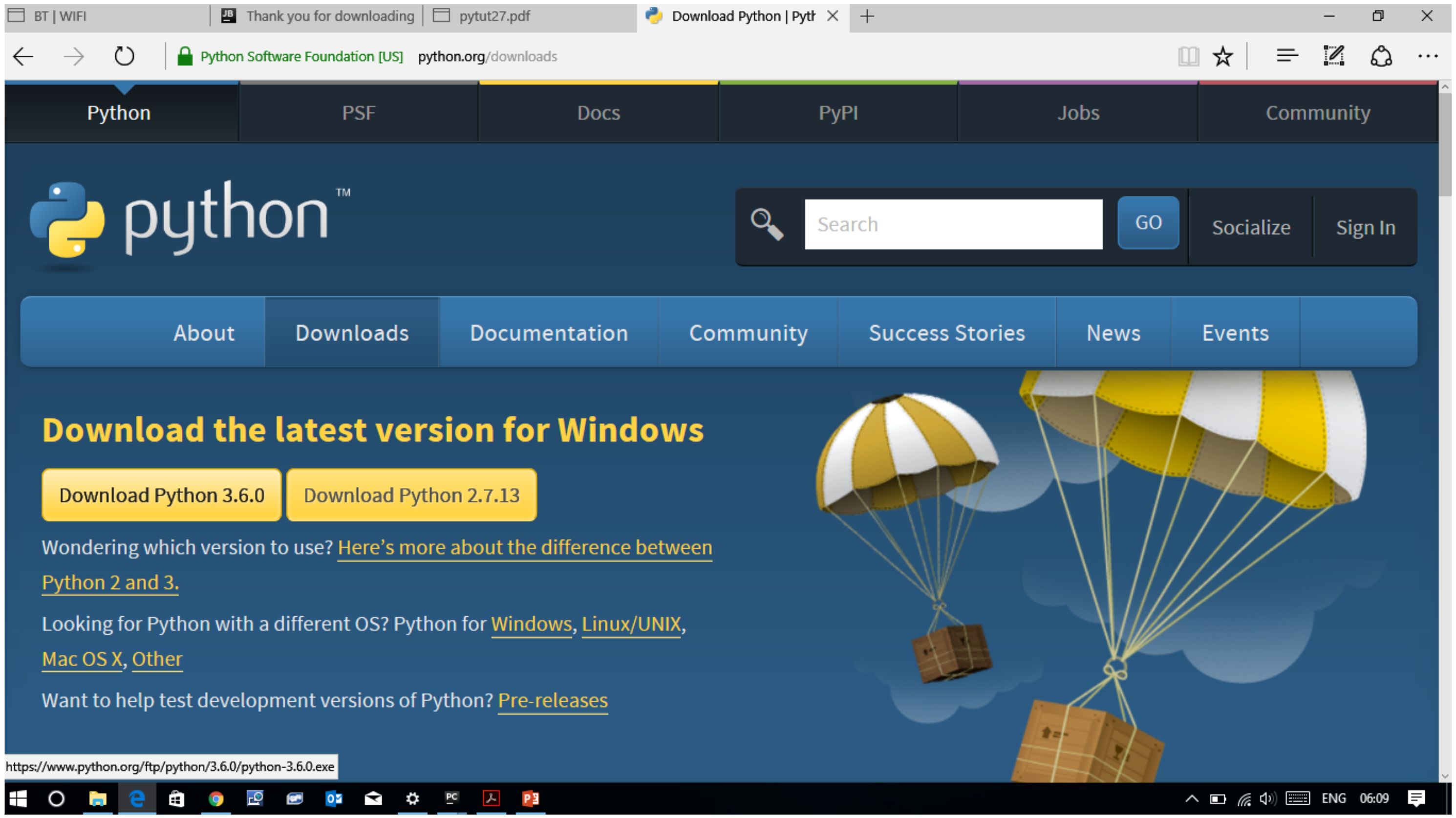
- **CPython**
  The original and the most common version of Python

- **Jython**
  Python interpreter that runs on Java JVM and offers great java integration

- **Iron Python**
  Python version for .NET MS framework integration

- **Others**
  Pypy, Rpython, Python Compilers

# Python Installation



Download the latest version for Windows

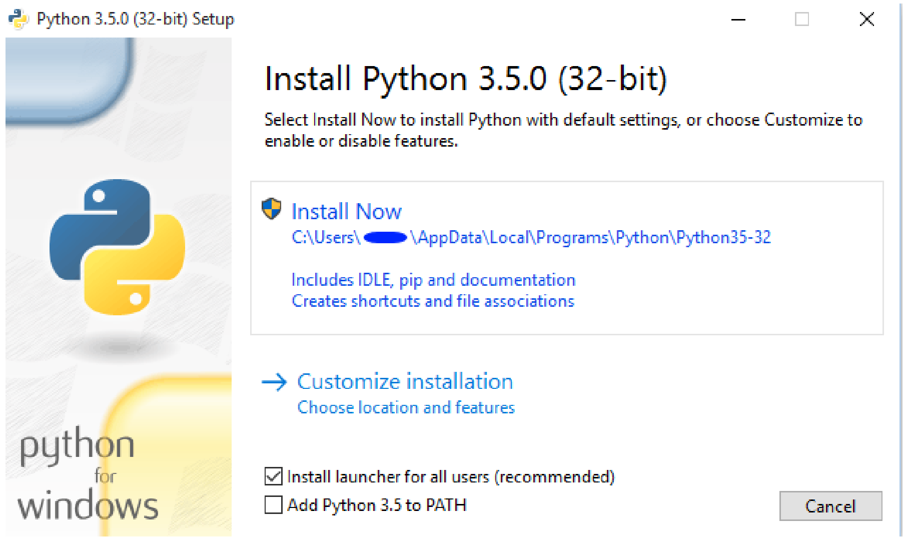
Wondering which version to use? Here's more about the difference between Python 2 and 3.

Looking for Python with a different OS? Python for Windows, Linux/UNIX, Mac OS X, Other

Want to help test development versions of Python? Pre-releases

# Python Installation

- Copy python kit on local disk (*python-3.5.2.exe*)
- Start Installation
- Install for all users – optional
- Click Add Python to PATH
- Click "Install Now"
- Finish Installation
- Run Windows "*cmd*"
- Run "*python*"


Python 3.5.0 (32-bit) Setup

Install Python 3.5.0 (32-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\[redacted]\AppData\Local\Programs\Python\Python35-32

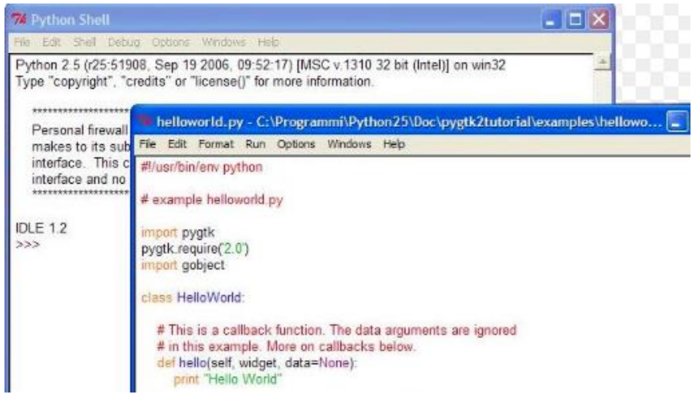
Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☐ Add Python 3.5 to PATH

# IDLE – Simple IDE

- Tool used to make writing programs easier
- Provides both an interactive mode ***and*** a script mode
- Syntax Highlighting
- Code auto-complete
- Integrated Debugger
- Included in Python installation


Python Shell
File Edit Shell Debug Options Windows Help
helloworld.py - C:\Programmi\Python25\Doc\pygtk2tutorial\examples\hellowo...
File Edit Format Run Options Windows Help

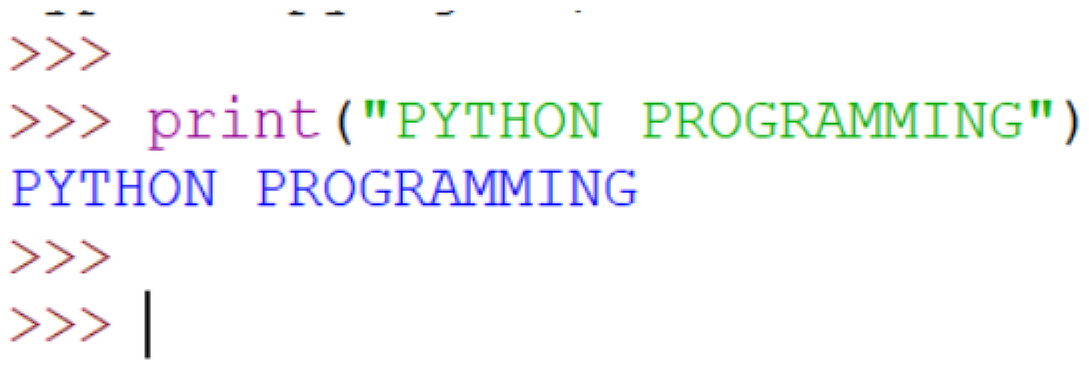
```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    ********************
    Personal firewall
    makes to its sub
    interface.  This c
    interface and no
    ********************

IDLE 1.2
>>>
```

```
#!/usr/bin/env python

# example helloworld.py

import pygtk
pygtk.require('2.0')
import gobject

class HelloWorld:

    # This is a callback function. The data arguments are ignored
    # in this example. More on callbacks below.
    def hello(self, widget, data=None):
        print "Hello World"
```

# Writing a Program

Type: `print ("PYTHON PROGRAMMING")`

Congratulations! This is a very simple program you have just coded, and should look like this:



```
>>>
>>> print("PYTHON PROGRAMMING")
PYTHON PROGRAMMING
>>>
>>> |
```

# Using Script Mode

Script mode allows you to write, edit, load and save your coding.  This can be accessed via:

*File > New File or Open*

If the same statement is entered:

```
print (“PYTHON PROGRAMMING”)
```

you will find that nothing happens as it did in the interactive mode.  This ‘script mode’ simply provides space to list statements for the computer to execute at a later time

> Click run module to run the script

# Python Basics

- ✓ Statements, Blocks, Comments
- ✓ Expressions
- ✓ Variables and Data Types
- ✓ Operators
- ✓ Lists
- ✓ Conditional Expressions

## Statements

What is a statement ?

The smallest standalone unit of programming code. A script is formed by a sequence of statements.

- One statement usually takes one line

```
new_a = old_a + b
```

- Escape character “\” can be used to continue the statement on the second line

```
new_a = old_a \
     + b * c
```

## Statements

- Semi-column character “;” can be used to put multiple statements on the same line

```
new_a = "my Text”
print (new_a)
```

- Empty lines are ignored by Python. They can be used to enhance code readability

- A line starting with hashtag character “#” is ignored and it is considered a comment. They can be used to explain the code.

```
# This is a code for my school
#
```

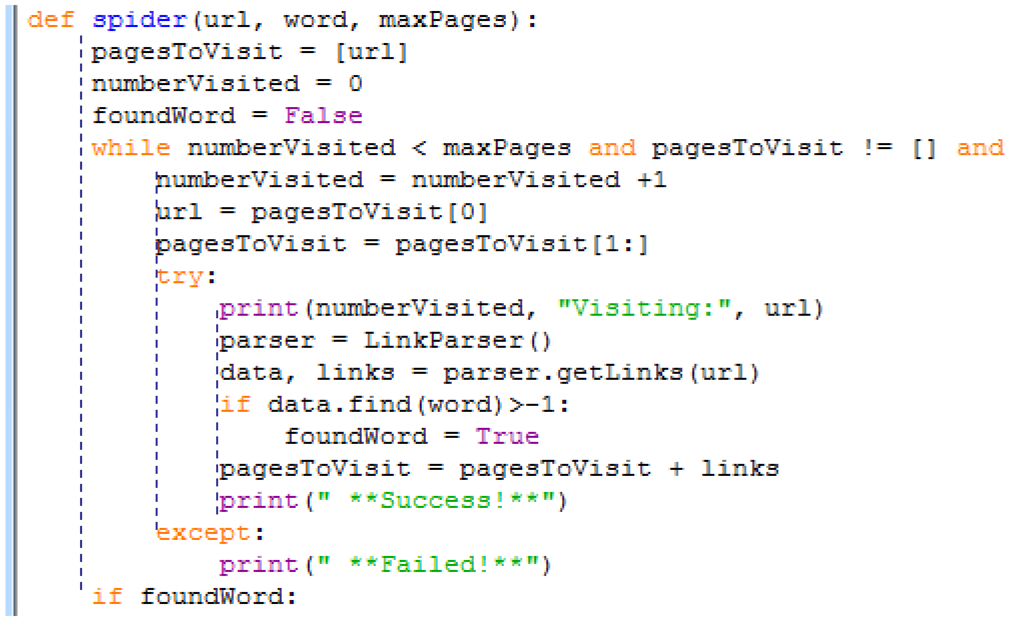
## Blocks

What is a block ?

- A block is consisting from one or more consecutive lines indented by the same amount. Indenting sets lines not only visually, but logically, too. Together, they form a single unit of execution
- Python is not using brackets “{ }“ or statements BEGIN / END to mark execution blocks
- Blocks are used in IF conditions, functions, classes and modules.
- A single Python command in shell acts like a block

# Blocks

- Python is not using brackets “{ }“ or statements BEGIN / END to mark execution blocks
- All statements with the same shift from left are in the same block



```
def spider(url, word, maxPages):
    pagesToVisit = [url]
    numberVisited = 0
    foundWord = False
    while numberVisited < maxPages and pagesToVisit != [] and
        numberVisited = numberVisited +1
        url = pagesToVisit[0]
        pagesToVisit = pagesToVisit[1:]
        try:
            print(numberVisited, "Visiting:", url)
            parser = LinkParser()
            data, links = parser.getLinks(url)
            if data.find(word)>-1:
                foundWord = True
            pagesToVisit = pagesToVisit + links
            print(" **Success!**")
        except:
            print(" **Failed!**")
    if foundWord:
```

# Creating Variables

- Before you use a variable you have to create it:

```
name = “Larry”
```

- This line is called an *assignment statement*. It creates a variable called name and assigns it a value so that it references the string `“Larry”`
- Once the variable is created, it can be used in the program: `print(name)`
- A variable data type does not have to be declared before (like in other languages) – in this case type string

## input() Function

Using the input function to get data from keyboard in a script. The execution will stop until the user input the data.

```
name = input(“Hi. What’s your name? “)
```

As previously shown, `name` is created and a value is assigned to it. However, this time the right side of the assignment statement is a call to the function `input()`.

# Variable Assignment

- Simple assignment (variable = static value)
- Using the input function (from keyboard)
- Multiple assignment

```
a = b = 0
a, b = 5, 7.1
```

- Expression assignment – Use an expression

```
a = b + 2 * sqrt(2)
```

- Use ‘del’ to un-assign (release) a variable

## Data Types

| Data Type | Type | Example |
|---|---|---|
| NUMBER | INT, FLOAT COMPLEX | n = 100, x = 1.3 |
| STRING | CHAR | name = “Larry” |
| BOOLEAN | LOGICAL | x = True , v1=False |
| NONE | NULL | a = None |
| SEQUENCE | TUPLE, LIST, SET, DICT | list = [] |

## Numbers

- INT – long signed integer

```
a = 100, b = -3
```

- FLOAT – decimal or real values

```
a = 2.53, b = -7.1, c = 0.5e+7
```

- Complex – real and imaginary (math)

```
a = 1.5+2j, b = -7j
```

# Numeric Operators

- Addition and Subtraction : (+), (-)
- Multiplication (*)
- Division (/) – it always returns a float
- Floor division (//)
- Modulus (%) - returns the reminder
- Power operator (**)

The order of operations is important:  (), **, */, +-

```
z = (3+1)**2/4 + 0.5*2
```

# Augment Numeric Operators

| Operator | Example | Is Equivalent To |
|---|---|---|
| *= | x *= 5 | x = x * 5 |
| /= | x /= 5 | x = x / 5 |
| %= | x %= 5 | x = x % 5 |
| += | x += 5 | x = x + 5 |
| -= | x -= 5 | x = x - 5 |

Note: There is no incremental operator in Python: x++

## Strings

- Strings are text values with characters
- Python 3 characters using UTF-8 encoding

  ASCII “a-z, A-Z” or 16/32 bit Hex "\u0394"

- Special characters ‘\’- escape, ‘\n’-new line, ‘\t…
- Strings values must be in quotes:

```
a = ‘This are single quotes’
 b = “double quotes are fine as well”

 c = '''even triple quotes if we need
        to split on more
        lines'''
```

# Sequence Array

- Sequence data type is Python is a generic name for a collection of elements (array)
- There are few data types Python

Tuple ()

List []

Set {}

Dictionary {}

String “ ” as a Sequence

# String Indexes

- Strings can be seen as a sequence of chars (array)
- Characters in a string are numbered with *indexes* starting from 0 to N-1

Example:

```
name = "P. Diddy"
```

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| character | P | . |  | D | i | d | d | y |

- Accessing an individual character of a string:

  *variable* [ *index* ] `or` *variable* [ *- index* ]

Example: `print (name, "starts with", name[0])`

```
P. Diddy starts with P
```

# String Operators

Concatenation: (+) or (' ‘)

'Hello' + 'World‘ = 'HelloWorld'

- Multiplication (*)

'Hi‘ * 3 = 'HiHiHi'

- Slice [:] on Index

```
>>> var1 = 'Some Text in Here‘
>>> var1[5]
'T‘
>>> var1[5:7]
'Te‘
>>> var1[-3]
'e‘
```

```
>>> var1[5:]
 'Text in Here'
>>> var1[:5]
'Some '
>>> var1[-3:]
'ere'
```

# Tuple and List

- A collection of elements separated by (,)
- Each element can have any Python data-type
- An element can be another tuple/list – multi-dimensional
- It can contain duplicate elements/values

| **Index 0** | **1** | **2** | **3 …** |
|---|---|---|---|
| 12.4 | 'Text value' | True | (1,2,3,4) |

## Difference between Lists and Tuples

- List elements can be changed (mutable)
- Tuple elements cannot be changed (immutable or fixed lists)

# Tuple and List

Lists notation [1,2,3] - square brackets

Tuple notation (1,2,3) – round brackets

Examples

v2 = ('red', 'green', 'yellow', 'blue')

The values can be mixed (number, strings, etc)

>>> v3=(1, 2, 'uau', None, -7.5, False)

Multi-dimensional (matrix)

v3 = ((1,2), (7,-1), (5,0));   v4 = [(‘A’,12), (‘B’,17)]

# Tuple & Lists Operators

Concatenation: (+)

```
(1,5,7) + (2,9,1001) = (1, 5, 7, 2, 9, 1001)
```

Multiplication (*)

```
(1,5,'*')*3 = (1,5,'*',1,5,'*',1,5,'*')
```

Slice [:] on Index

```
v = (1,2,'uau',None,-7.5,False)
```

```
>>> v[2]                    >>> v[-2]
'uau‘                       -7.5
>>> v[2:4]
('uau‘, None)
```

For multi-dimensional use Slice [:][:] Index `>>> v4[1][0]`

# List Assign Element

An element can be changed from its initial value, using his index [0 .. N-1], to a new value/object

```
list1 = [1, 4, -5, 100]

list1[2] = 51
```

Then, list1 = [1, 4, 51, 100]

This does **not** work for a tuple, **nor** for a string

```
>>> s[8]='A'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

## Sets

- A collection of **unique** elements separated by (,)
- Can contain mixed data-types
- Sets cannot be multi-dimensional (contain other sets)
- Sets can be modified (Mutable)

Sets notation using {} brackets

```
>>> A = { 'A', 'B', 'C'}
>>> B = {1, 77, -10, 20}
```

## Set Operators

Union ( | )

Intersection (&)

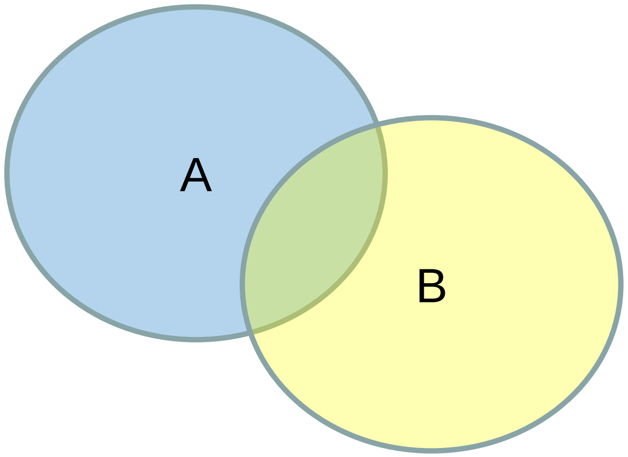
Minus (-)

A={2, 7, 9, 12},  B={7, 12, 900}

A | B  = {2, 900, 7, 9, 12}

A & B = {12, 7}

A – B =  {9, 2}



*Note: there are **no** index A[0]? , no slice [:], no concatenation (+), no multiplication (*) operators for sets*

# Dictionary

- A list of key : value pairs separated by (,)
- Key must be unique and immutable within dictionary
- Values can be any Python data type
- Dictionaries can be updated based on key value

Dictionary syntax is using {} brackets

```
Dict =
{"Name" : "John", "Age":27, "City" : "Brisbane", "Commission" : True}
Oth =
{"Name" : "Matt", "Age":35, "City" : "Adelaide", "Rates":[0, 1.25, 7.5]}
```

# Dictionary Operators

- Access Dictionary Elements by [Key]

>>> Di["Name"]  -> return  'John‘  (return None if key is not present)

>>> Di.get['key', 'Rep']  -  returns the value for key in dictionary
(if key is not present return ‘Rep’)

- Update Dictionary Value for a Key

  Di["Name"] = "Jonny“

Di = {"Name" : "Jonny", "Age":27, "City" : "Brisbane", "Commission" : True}

*Note: there are* ***no*** *index A[0]? , no slice [:], no concatenation (+), no multiplication (*) operators for dictionaries*

## Comparison Operators

- Logical expressions are often created by comparing values. You can compare values using ***comparison operators***

| Operator | Meaning | Sample Condition | Evaluates To |
|---|---|---|---|
| == | equal to | 5 == 5 | True |
| != | not equal to | 8 != 5 | True |
| > | greater than | 3 > 10 | False |
| < | less than | 5 < 8 | True |
| >= | greater than or equal to | 5 >= 10 | False |
| <= | less than or equal to | 5 <= 5 | True |
| IS | equal to | 5 is 5 | True |

# Element Comparison

- For strings, lists and sets: there is a special operator to test if one element belongs in the list/string

IN (NOT IN)

```
>>> 'p' in 'Aeroplane'
True
>>> 3 not in (1, 4, 77, 3, 21)
False
```

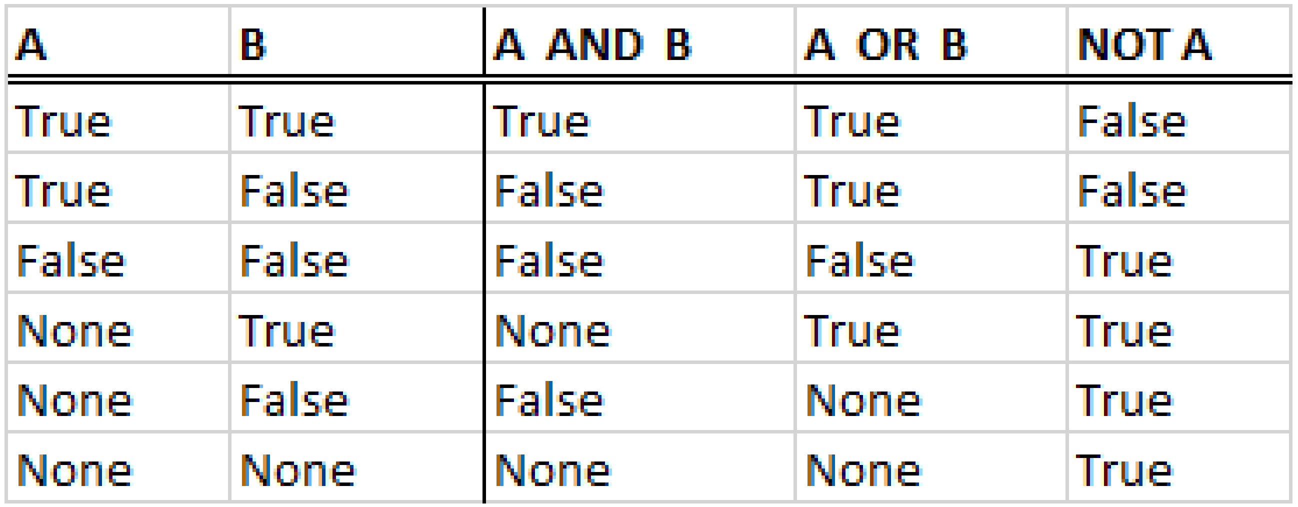
For dictionaries, the IN operators works for the key value

```
>>> "zero" in {"zero":0, "one":1, "two":2}
True
```

## Logical Expression

- Comparisons can be combined together in more complex logical expressions using logical operators: NOT, AND, OR

Example: np > 3 AND T != 0 OR True



| A | B | A AND B | A OR B | NOT A |
|---|---|---|---|---|
| True | True | True | True | False |
| True | False | False | True | False |
| False | False | False | False | True |
| None | True | None | True | True |
| None | False | False | None | True |
| None | None | None | None | True |

# Flow Control and Functions

- ✓ IF Statement
- ✓ Functions
- ✓ String Functions
- ✓ List Functions
- ✓ Loops
- ✓ Iterators and Generators

# IF Statement

Control flow statement IF is used to execute a block only when one condition is True

Single Line: if <condition> : <action>

Multiple Lines:
```
if <condition> :
        <action block>
        ….
```

IF ELSE – elocution control if condition is not True

```
if <condition> :
        <action block>
else:
        <action block>
```

# IF Statement

Example:

```
if x < 0:
        x = 0
        print ('Negative changed to zero')
else:
       print('Value: ', x)
```

IF Condition

- It is a logical expression that evaluates True or False
- Empty (), [],{} always evaluates False

```
if () : print('can\'t see this ! ')
```

## ELIF

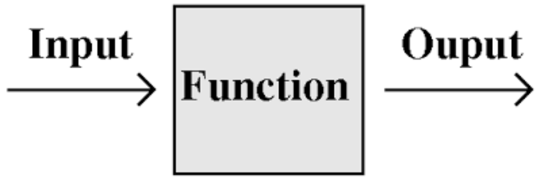
- There is no CASE or SWITCH statement in Python
- For multi-choice control flow statement:
    - Nested IF statements
    - ELIF clause in IF statement

Example:

```
if x < 0:
     x = 0 ;
elif x == 0:
    print ('x is zero')
elif x < 10:
    print ('x is a digit')
elif x < 100:
    print ('value:', x)
```

# Functions

- Functions take one or more arguments and return only one value/object (or None)



- Functions can be stand-alone ex: type(S)
- Function can be member of a class, in which case the function is called a “method”
- Python has a large library of built-in functions: Numeric, String, Conversion, functions for tuples and lists
- User-defined functions can be created as well.

# Useful String Methods

| Method | Description |
|---|---|
| `Len(string)` | Returns the length of string in characters |
| `s.lower(), s.upper()` | Returns the lowercase version of the string |
| `s.find/s.rfind` | Search for a substring and returns index |
| `s.capitalise()` | Returns a new string where the first letter is capitalised and the rest are lowercase |
| `s.split(char)` | Return a list of words - Split string into a list, default ‘ ' separator if not defined |
| `s.strip()` | Returns a string where all the white space (tabs, spaces, and newlines) at the beginning and end is removed |
| `s.replace(`***`old, new`***` [`***`,n`***`])` | Returns a string where occurrences of the string ***`old`*** are replaced with the string ***`new`***. The optional ***`max`*** limits the number of replacements |

# Conversion Functions

| Method | Description |
|---|---|
| `str(int,float)` | Converts a number into a string |
| `int(str), float(str)` | Converts a string into a int or float |
| `str(tuple, list…)` | Converts a tuple to its string representation (print) |
| `tuple(str)`<br>`list(str)`<br>`set(str)` | Creates a tuple/list/set from the letters of string argument |

Example:

```
>>> s='This is the string'
>>> list(s)
['T', 'h', 'i', 's', ' ', 'i', 's', ' ', 't', 'h', 'e', ' ', 's', 't', 'r', 'i', 'n', 'g']
```

## Common Functions

Applicable to all: tuple, list, set & dictionary

- len([,]) - return number of elements
- x.clear() – delete all elements (*tuple)
- del (s) – delete the object

Aggregation function:

- sum([,]) - sum of all elements of list/tuple/set
- max([,]) - max element from list/tuple/set
- min([,]) - min element from list/tuple/set

-- they only work if all elements are numeric

## Specific functions for Lists

Find element

- l.count (obj) - count how many times obj is in the list
- l.index (obj) - find the first occurrence of obj in the list

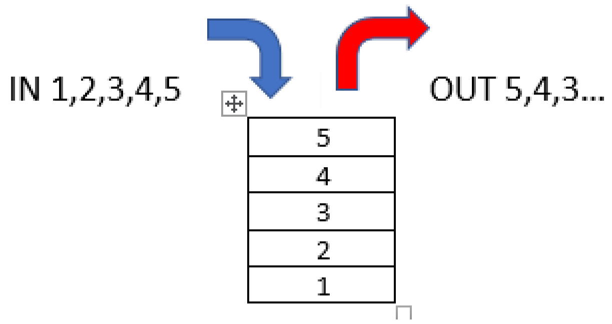
Insert List Elements

- l.insert (idx, obj) - insert an object in list (at index pos)

Delete Elements

- del l[:] - remove a specific element by index/range
- l.remove (obj) - remove the obj from the list (from left)

# Specific functions for Lists



LIFO (Last-In-First-Out)

- l.append (obj) - append an object to the list (right)
- l.pop() - remove and return the last added object

FIFO (First-In-First-Out) using deque module

- l.reverse() - reverse the order of a list

# Dictionary Specific Methods

Add a pair key : value to a dictionary

```
>>> d.update({'key': val})  or  d[‘key’] = val
```

Remove a key from a dictionary : del d['key']

Other dictionary methods

d.keys() - return a list of keys

d.values() - return a list of values

d.items() – coverts dictionary to list of (key : val) pairs

*Remember: you cannot use index S[0] for a dictionary (only S[‘key’])*

# Sets Specific Methods

Add a value into a Set

S.add( element ) – add a new element to a set

```
>>> B = {1, 77, -10, 20}
>>> B.add (3.14)
```

Remove elements

S.remove( element ) – remove an element from set

```
    B.remove(77)
```

S.discard( element ) – silent remove

There are functions available for set operators.

## Sorting Data

Sorting data is one of the most common tasks for data processing on computers.

Sort values using a function

sorted(tuple | list | set [, reverse=True])

- return a list with sorted values

or a method: l.sort () - This method is applicable only for lists

Example:

g = (1, 0.25, 1000, 50, 7)

>>> sorted(g) - will return: [0.25, 1, 7, 50, 1000]

>>> sorted(g, reverse=True) – will return : [1000, 50, 7, 1, 0.25]

# Complex Sorting

- Sorting works good when all values from the list are of the same data type (numeric or alpha-numeric)

  *TypeError: unorderable types: str() < int()*

- x=[0, 1, 5.5, True, None, 77, 'String']

Use custom sort key – convert all values to string

```
sorted(x, key=lambda x: str(x))
    [0, 1, 5.5, 77, None, 'String', True]
```

Complex sorting:

- Sort based on computed value or substring
- Multi-dimensional list sorting : key=lambda p: (p[0],p[1])

## Range

- Range is a special function to generate sequence of immutable numbers
- Ranges are generated sequences (not lists)

range([start], stop, [step])

Parameters must be integers

start - start value (Default = 0)

stop - last value

step - increment value (Default = 1)

Example: range (0, 10, 1)

## Range

- Range is very useful in loops for iteration
- Elements can be addressed using index R[0..N-1]

Example of range functions:

range(0,10) , generates [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

R = range(-2,30,5),  generates [-2, 3, 8, 13, 18, 23, 28]

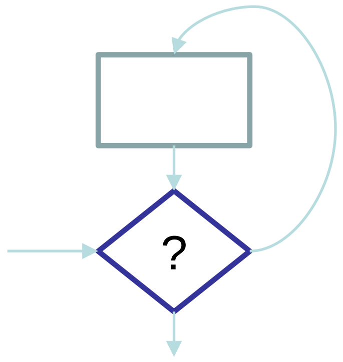
R[5] is 23

you can use tuple(range()), list(range()), set(range())
to convert a range to a tuple, list, set

# Loops

What are loops ?

- A sequence of instructions that are continually repeated until a certain condition is met
- In Python there are 2 types of Loops
  1. WHILE Loop
  2. FOR Loop



Loop Control: (except the exit condition)

- Break
- Continue
- Pass

# While Loop

```
while <condition> :                # while condition is True
    <action stmt's>
[else:]                            # if condition is False
    <action>
```

Example

```
while i < 10:
        print (i)
        i+=1
else:
        print (" It is bigger !")
```

# FOR Loop

```
for <variable>  in  <sequence>:
      <action stmt’s>
```

Examples:

| | |
|---|---|
| for element in [1, 2, 3]:<br>        print(element) | Sequence from LIST |
| for char in "123 string":<br>        print(char) | Sequence from String |
| for key in {'one':1, 'two':2}:<br>        print(key) | Sequence from Dictionary Keys |

## Using the break Statement to Exit a Loop

- In this plan, we necessitate to check if the magic number in the range of 0 to 101 using for loop,
- Detect when the program sees the number 26 stop the loop using break statement.

```
magicNumber=26
for n in range (101):
    if n is magicNumber:
        print(n, "is the magic number")
        break
    else:

        print(n)
```

## Using the continue Statement to Jump Back to the Top of a Loop

```
while True:
    count += 1
    print(count, "is the count number")
    if count == 5:
        continue
```

```
1 is the count number
2 is the count number
3 is the count number
4 is the count number
6 is the count number
7 is the count number
8 is the count number
9 is the count number
10 is the count number
11 is the count number
12 is the count number
```

# Iterators

Iterators can be created from any type of sequence used in loops, like: string, tuple, list, set, dictionary, range and more…

for var in <sequence>:

The sequence is an “iterable”.“For” loop automatically apply the iteration operator to a sequence :

it = iter(seq), it.__next__() for all elements

Iterators can be iterated as many times as we like.

## Iterators

- Technically speaking, Python iterator object must implement two special methods, __iter__() and __next__(), collectively called the iterator protocol

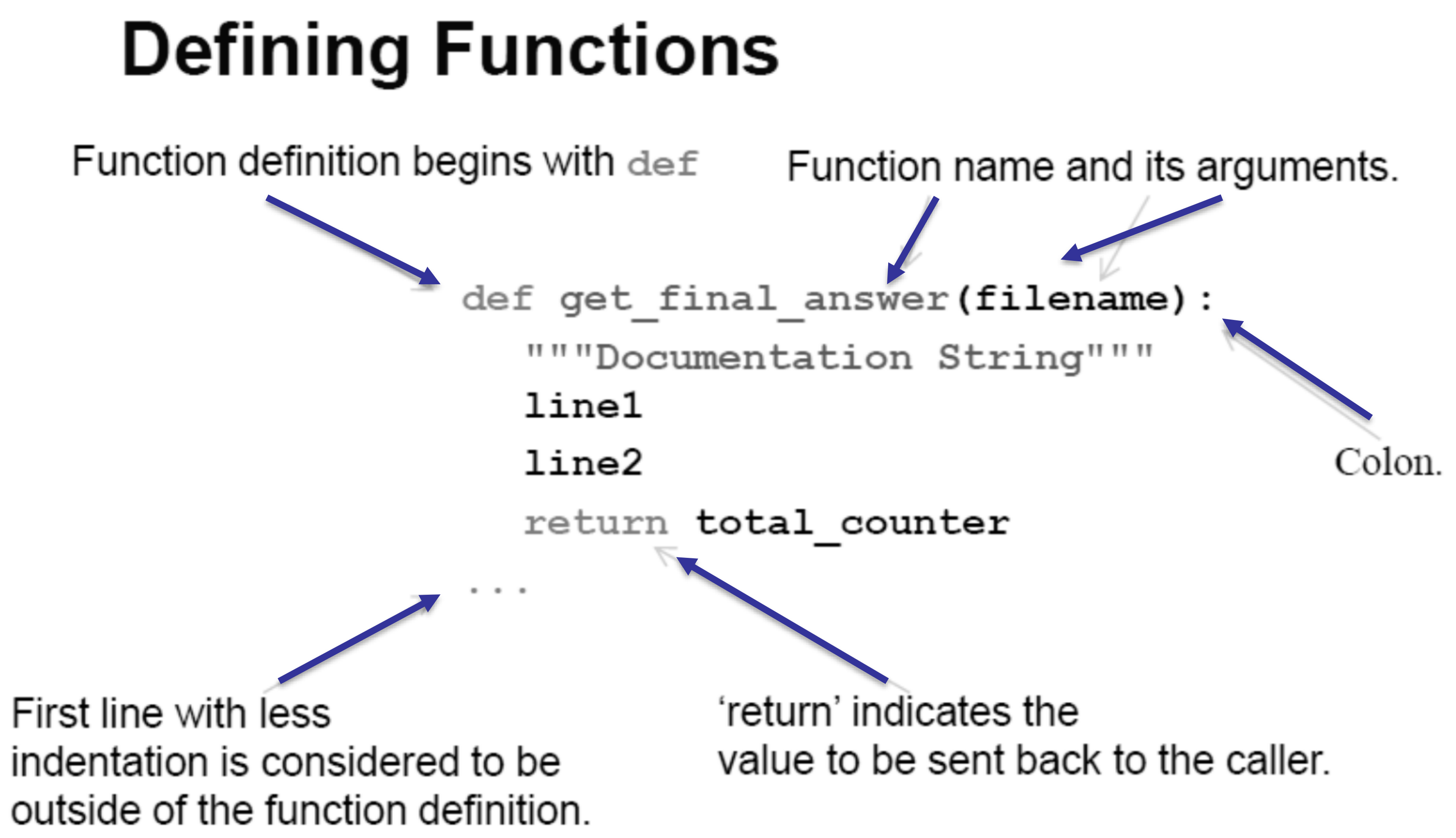
```
my_list = [1,2,3]
my_iterator = iter(my_list)
while True:
    print(next(my_iterator))
```

```
my_list = [1,2,3]
my_iterator = iter(my_list)
while True:
  print(my_iterator.__next__())
```

# Python Programming

- ✓ User Defined Functions
- ✓ Modules
- ✓ Random Numbers
- ✓ Date and Time
- ✓ Exceptions
- ✓ Regular Expressions

# Functions in Python



## Defining Functions

Function definition begins with `def`

Function name and its arguments.

```
def get_final_answer(filename):
    """Documentation String"""
    line1
    line2
    return total_counter
...
```

Colon.

First line with less indentation is considered to be outside of the function definition.

‘return’ indicates the value to be sent back to the caller.

**No declaration of types of arguments or result**

## Function Example

```
def area (x,y):
        "calculates area of a rectangle"
        sf=x*y
        return sf
```

Function call:

area(2,7)  returns 14;    area(3,8) returns 24

To display the function description area.__doc__
'calculates area of a rectangle'

# Optional Parameters

- A function can have few optional parameters
- Parameters have specified default values

```
def area(x,y = 1):
      "calculates area of a rectangle"
      sf=x*y
      return sf
```

- In case "y" is not provided in the function call

it will have default value 1

>>> area(3)  returns 3

## Parameter value vs. pointer

- Normally, a function use input parameters as values
- Parameter value cannot be modified within function

If we pass a list (pointer) instead of a value:

```
def alter_str (list1):
    list1[2]='Moddify Now'
    return list1
```

- Then, the function will be able to modify list elements
- In this case it is not necessary to declare list as global

## Example

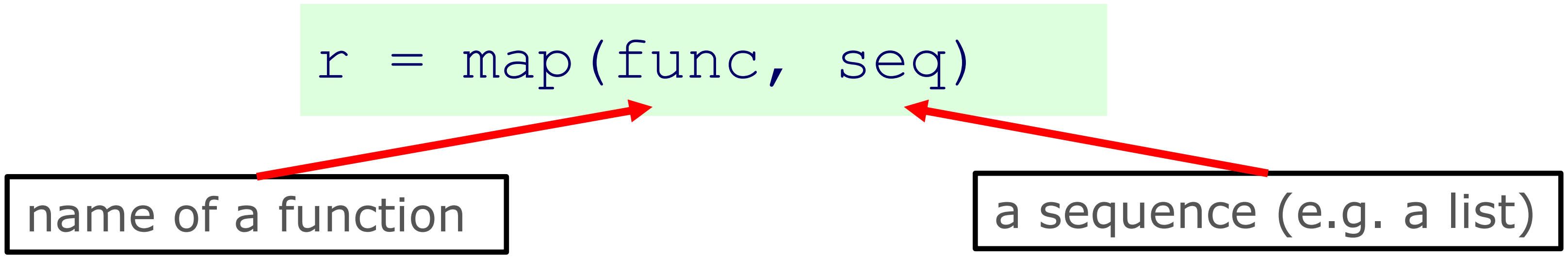
```
def get_gender(sex='Unknown'):
    if sex is 'm':
        sex="Male"
    elif sex is 'f':
        sex="Female‟
     print(sex)
get_gender('m')
get_gender('f')
get_gender( )
```

```
>>> def myfun(x): return x*3

>>> def apply(q, x):
     return q(x)

>>> apply(myfun, 7)
```

# The map() Function

- The advantage of the lambda operator can be seen when it is used in combination with the map() function.

map() is a function which takes two arguments:



```
r = map(func, seq)
```

```
def fahrenheit(T):
   return ((float(9)/5)*T + 32)
def celsius(T):
   return (float(5)/9)*(T-32)
temperatures = (36.5, 37, 37.5, 38, 39)
F = map(fahrenheit, temperatures)
C = map(celsius, F)
temperatures_in_Fahrenheit = list(map(fahrenheit, temperatures))
temperatures_in_Celsius = list(map(celsius,
temperatures_in_Fahrenheit))
print(temperatures_in_Fahrenheit)
```

```
[97.7, 98.60000000000001, 99.5, 100.4, 102.2]
```

# The map() Function

- In the last example we haven't used lambda. By using lambda, we wouldn't have had to define and name the functions fahrenheit() and celsius(). You can see this in the following interactive session:

```
C = [39.2, 36.5, 37.3, 38, 37.8]
F = list(map(lambda x: (float(9)/5)*x + 32, C))
print(F)
```

- The output:

```
[102.56, 97.7, 99.14, 100.4, 100.03999999999999]
```

## Modules and Packages

- Python *module*— a python source code file containing variables, functions and classes/objects.
- Each module has its own namespace – which is usually the module name.

Use import command to import a module

```
import module1 [, module2]
```

- Python *package* – multiple modules can be grouped in a package

## Random numbers

- Random numbers are useful in programming for statistics, science and games
- For using random numbers – Python module “random” (standard library)

```
import random

# random float between 0.0 and 1.0
>>> random.random()
0.1359922746826534
# random int
>>> random.randint(0,10000)
8250
# random char (from string)
random.choice(0123..abcd…)
```

# Date and Time

Python modules for date and time provide support for retrieving system date and time, operations with dates, calendar and time-zones.

- Date and time tuple – date structure with elements in the form of:

  (tm_year, tm_mon, tm_mday, tm_hour, tm_min, tm_sec, tm_wday, tm_yday, tm_isdst)

- Modules datetime, time and calendar

```
>>> import datetime
>>> datetime.date.today()
 datetime.date(2017, 4, 25)      method returning object today
```

# Date and Time

Date and time object:

d = datetime.date.today()

t = datetime.datetime.now()

| Method | Description |
|---|---|
| `t.timetuple()` | Method that return the time tuple, including hour, minutes and seconds |
| `t.day, t.hour` | Object attributes (year, month, day, hour…) |
| `t.now()` | Method returns a tuple with current time (microsec) |
| `t.utcnow()` | Timezone UTC tuple |
| t.strftime(“ format “) | Method returns a string with date formated |

# Operations with Dates

## Dates can be added, subtracted and compared

```
Assign Date
d1=datetime.date(2016, 3, 23)
d1=datetime.datetime(2017, 12, 11, 5, 10, 55)

Replace Method: d2 = d1.replace(year=2009)

Add a specific time interval (specific number of days, hours)
d1 + datetime.timedelta(hours=200) = datetime.date(2015, 3, 31)
d2 - datetime.timedelta(days=15) = datetime.date(2017, 3, 7)

Add and Substract Dates
d2=datetime.date.today()
d1-d2 = datetime.timedelta(-730)

Compare Dates
if d2 > d1:
    d1 = datetime.date.today()
```

# Exception Handling

- **Exceptions are Python objects**
- More specific kinds of errors are subclasses of the general Error class.

- **You use the following forms to interact with them:**
- try
- except
- *else*
- finally

## Handling Exceptions

- When Python runs into an error, an error message is displayed. This is known as raising an **exception**

- If no actions are carried out, Python halts what it's doing and displays an error message detailing the exception

E.g.:

```
ValueError: could not convert string to float: Hi!
```

- Using Python's exception handling functionality, you can intercept and handle exceptions so that your program doesn't end abruptly

# Using a Try Statement with an Except Clause

- By using a `try` statement, you section off some code that could potentially raise an exception
- Then you write an except clause with a block of statements that are executed

```
try:
   < code that potentially could throw exception1 >
    ...
except(exception1):
   < execute in case of exception 1 or 2>
   ...
else:
   < execute in case there was no exception1 >
```

# Specifying an Exception Type

| Exception Type | Description |
|---|---|
| `IOError` | Raised when an I/O operation fails, such as when an attempt is made to open a non-existent file in read mode |
| `IndexError` | Raised when a sequence is indexed with a number of non-existent element |
| `KeyError` | Raised when a dictionary key is not found |
| `NameError` | Raised when a name (e.g. of a variable or function) is not found |
| `SyntaxError` | Raised when a syntax error is encountered |
| `TypeError` | Raised when a built-in operation or function is applied to an object of inappropriate type |
| `ValueError` | Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value |
| `ZeroDivisionError` | Raised when the second argument of a division or modulo operation is zero |

## Getting an Exception’s Argument

- When an exception occurs, it may have an associated value, the exception’s **argument**
- The argument is usually an official message from Python describing the exception
- You can receive the argument if you specify a variable after the exception type, preceded by the keyword `as`

•

```
>>> def divide(x, y):
     try:
          •result = x / y
     except ZeroDivisionError:
           print "division by zero!"
     else: print "result is", result
     finally:
          •print "executing finally clause"

>>> divide(2, 1) result is 2
executing finally clause

>>> divide(2, 0) division by
zero! executing finally
clause

>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
File "<stdin>", line 1, in ?
File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

## Regular Expression

Searching for text strings using patterns is called regular expressions (REGEX).

- Sometimes searching for specific strings or substrings is not satisfactory in programming

Example: email address in form of

name@server.com

Python module “re” provides support for regular expressions

## Regular Expression

Escape char problem in Python

- Because back-slash char “\” is used both as escape char in Python and in REGEX at the same time
- There is a potentially issue of using “\” in REGEX patterns

Solution: Use raw strings instead of strings for REGEX patterns. Raw strings will ignore “\”

Example: r"this is \' a raw string"

# Regular Expression

REGEX Pattern characters

```
. - Match any single character (except \n)
^ - Match the start of string
$ - match the end of string
+ - one or more char before
? - zero or one char before
* - zero or more char before
\d - match decimal string
\w - Match any char of a-z or A-Z or 0-9 and underline (_)
\W - Match any special character !%$# (with escape \)
\s - Match any space character
\n - match new line
[a-e] - match a single character between 'a' and 'e'
Any other character (ex. c, B, 7) match only themselves
```

## Regular Expression

To search for a pattern anywhere in the string

```
r1 = re.search(r'pattern', string, flags = 0)
```

- return an object (ref. r1)
- if flags = re.I (seach case-insensitive)
- you can use r1.groups() - to return a tuple with all matches

## Object-oriented Programming

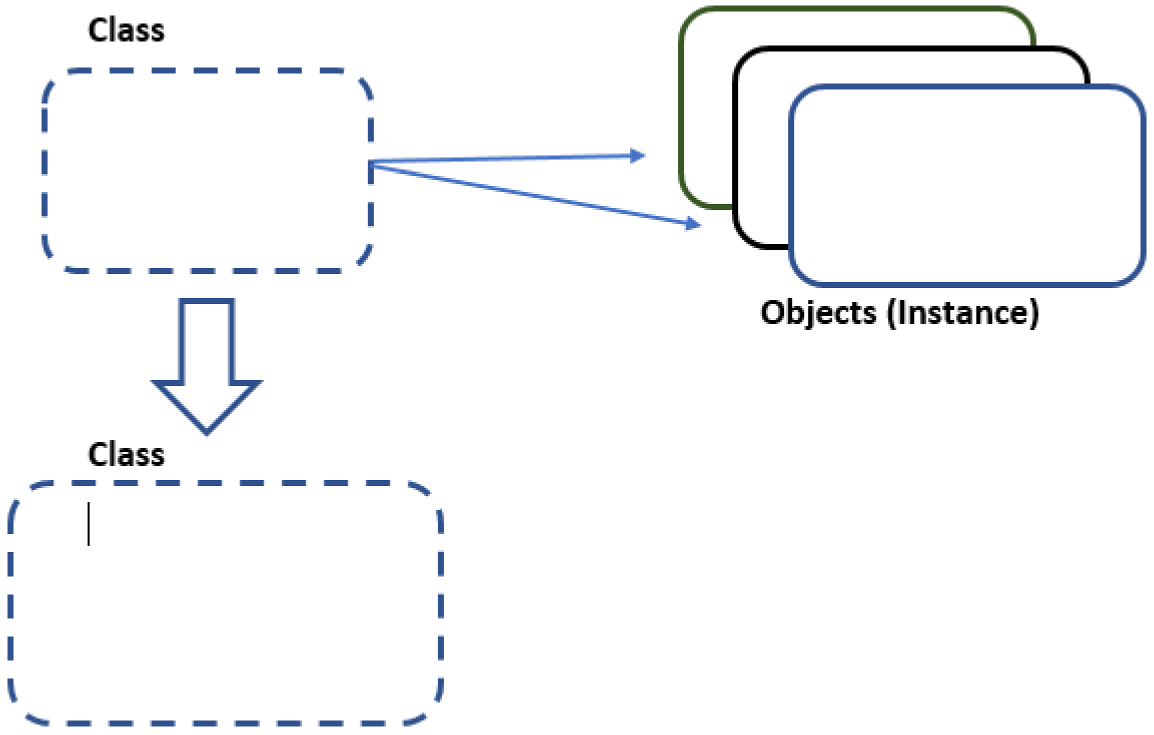
- **Object-Oriented Programming (OOP)** is a modern methodology that has been embraced by the software industry and is used in the creation of the majority of new, commercial software
- The basic building block in OOP is the software object - often just referred to as an object
- Also the concept of “encapsulation” , where complexity is hidden (encapsulated) in a container.

## Understanding Object-Oriented Basics

- What you often want to represent in your programs are real-life objects
- Like real-life objects, software objects combine characteristics (attributes) and behaviours (methods)
- Objects are created (instantiated) from a definition called a class - programming code that can define attributes and methods

# Continued…

- Classes are like blueprints. It isn't an object, it is a design for one



## Creating Classes, Methods and Objects

- To build an object, you first need a blueprint or a class
- Classes almost always include methods, things that an object can do
- You can however, create a class without any methods

## Introducing the Simple Critter Program

- Here is a simple example of a class written in Python. In this instance, it can only do one thing: say hi.

```
// Simple Critter
// Demonstrates a basic class and object

  class Critter(object):
      """A virtual pet"""
      def talk(self):
          print ("Hi. I'm an instance of class Critter.")
  # main
  crit = Critter()

  crit.talk()
```

# Defining a Class

- The program starts with a class definition, the blueprint of the first critter. The first line of the program is the class header: `class Critter(object):`

- Python doesn’t require a class name to begin with a capital letter, however it is the standard convention

- Python bases the class on object, a fundamental, built-in type. (which is the same as: `class Critter:` )

- The docstring `“””A virtual pet”””` documents the class

## Defining a Method

- The last part of the class defines a method. It looks very much like a function:

```
def talk(self):
    print(“Hi. I’m an instance of class Critter.”)
```

- You’ll notice that `talk()` has one parameter, `self` (which it doesn’t happen to use).This parameter provides a way for a method to refer to the object itself

- It is a good practice to name class/object attributes with **nouns** and methods with **verbs**

# Instantiating an Object

- Takes just one line:

```
crit = Critter()
```

- This line creates a brand-new object of the `Critter` class and assigns it to the variable

- The new object will have it’s own copy of class code (attributes and methods) in a new memory location.

# Invoking a Method

- The new object has a method called `talk()`
- You can invoke this method just like any other, using dot notation:

```
    crit.talk()
```

- Class has its own namespace (.) and object methods create their own local scope.
- The line invokes the `talk()` method of the `crit` object. The method simply prints the string “`Hi. I’m an instance of class Critter.`”

## Using Constructors

- A special method that is automatically invoked right when a new object is created

  (object initializer)

- It is usually used to set up the initial attribute values of an object

- If a constructor is not defined, Python will use the default one to create the object

## Creating a Constructor

```
class Student:
    '''Generic class for all students'''
    stdCount = 0

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

- Special method __init__ is the constructor name
- Accepts parameters (except self), that will be used to create a new instance of object

```python
class Employee:
    'Common base class for all employees'
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
        print ("Total Employee %d" % Employee.empCount)

    def displayEmployee(self):
        print ("Name : ", self.name, ", Salary: ", self.salary)

emp1 = Employee("David", 2000)
emp2 = Employee("John", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
```

constructor method

# Creating a Constructor

- The first new piece of code in the class definition is the constructor method (also called the **initialisation** method):

```
 def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1
```

- Naming the method __init__, tells Python that this is the new constructor method

- As a constructor method, __init__ is automatically called by any newly created Employee object right after the object springs to life

## File I/O Operations

Python built-in functions to read/write files on disk

file object = open( file [, mode][, buffering])

buffering =

-1 (OS system) - default, 0 - no buffer, >0 lines

Types of files:

» Text files (lines \n)

» Binary (0101)

» Data (csv, XML, JSON)

## File I/O Operations

File object (file)

```
file f = open( file [, mode][, buffering])
```

file attributes: f.name, f.mode

File methods: open(), close()

read(count) - read bytes (system default size - all file)

write()

tell() - tell the file position

seek(offset, 0|1|2) - ex: seek(100) - 100'th byte from start

# Text Files

- Text specific encoding (the default being UTF-8)
- Line endings (\n on Unix, \r\n on Windows)

```
f = open('text.txt',r+)
```

```
f.readline()   # reads the first line
f.readlines()   # creates a list with each text line
f.writelines()  # write a list intro a text file
```

```
Example: listf = file1.readlines() ; len(listf) - # lines
>>> listf[0]
'The first argument is a string containing the filename. \n'
```

# Files and Directories

Working with files and directory for OS specific functions, like change directory, copy file…

```
import os        # import OS module
os.getcwd() - get the current directory
os.chdir('/../') - change the current directory
os.rename( "file1.py", "file2.py" )
os.remove("file1.py")    # remove a file
```

Verify if the path exists:

```
>>> os.path.exists('/tmp/dirname/filename.etc')
True
```

# Database

SQLite is included in Python standard distribution.

- Very easy to use
- Full compatible with SQL Standard
- Small Memory and disk footprint

MySQL is the most popular open-source database,

- Used in millions of installations
- Driver available for Python
- Multi-user and clustering
- Both open-source and commercial

Oracle is commercial top #1 database

- Enterprise grade performance and reliability
- Very feature reach and options
- Driver available for Python (cx_Oracle)

## SQL Language

What is SQL ?

- SQL – Structure Query Language
- The facto standard ISO/ANSI for querying data to relational databases
- Used for SQLite, My SQL, Microsoft SQL, Postgres SQL, Oracle and others

SQL Commands

- ➢ SQL Query – SELECT
- ➢ DML – Insert/Update/Delete
- ➢ DDL – Data Definition Language (Create Table..)

# SQL Language

## SQL Commands

SELECT *column1*, *column2, ...*   FROM *table_name*
*WHERE condition;*

INSERT INTO *table_name*   VALUES (*value1*, *value2*, *value3*, ...);

UPDATE *table_name*   SET *column1* = *value1*, *column2* = *value2*, ...
WHERE *condition*;

DELETE FROM *table_name*
WHERE *condition*;

## Database Connection

Connect to SQL Database (SQLite 3)

```
import sqlite3
connection = sqlite3.connect(‘test.db')

c = connection.cursor()      # open database cursor

c.execute('SELECT * FROM EMP_TABLE', t)
print(c.fetchone())
```

# Sqlite3 Lab

```
import sqlite3

conn = sqlite3.connect('test.db')
print ("Opened database successfully");

conn.execute('''CREATE TABLE COMPANY
       (ID INT PRIMARY KEY     NOT NULL,
       NAME           TEXT    NOT NULL,
       AGE            INT     NOT NULL,
       ADDRESS        CHAR(50),
       SALARY         REAL);''')
print ("Table created successfully");

conn.close()
```

# Thank You

Well Done & Good Luck

Keep in Touch

Giridhar276@gmail.com

+91-9550712233

https://www.linkedin.com/in/giridhar-sripathi/