Python Programming

Dive Deep

1

TechieNest
...Transforming Engineers to Technocrats

Index

Introduction

- Easy to learn – Beginner's programming language
- Cross-platform Interpreted Language
- Clean and Elegant Syntax
- Free and Open source
- Object-Oriented Programming Language
- Large Standard Library
- Integrated – Easily integrated with C, C++, JAVA, etc..
- Magic Methods

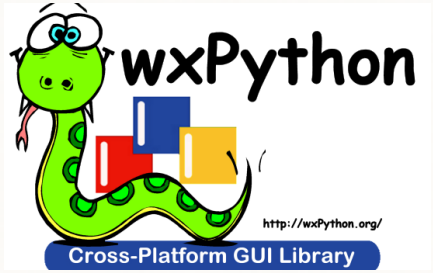
Monty Python are a British surreal comedy group who created the sketch comedy television show Monty Python's Flying Circus, which first aired on the BBC in 1969.

Application Areas

- **Web Development** – Django, Flask, Web2py, Bottle, Tornado

Application Areas

- **Graphical User Interfaces** – Kivy, PyQT, Tkinter, PyGUI, PySide, WxPython, pygame



Application Areas

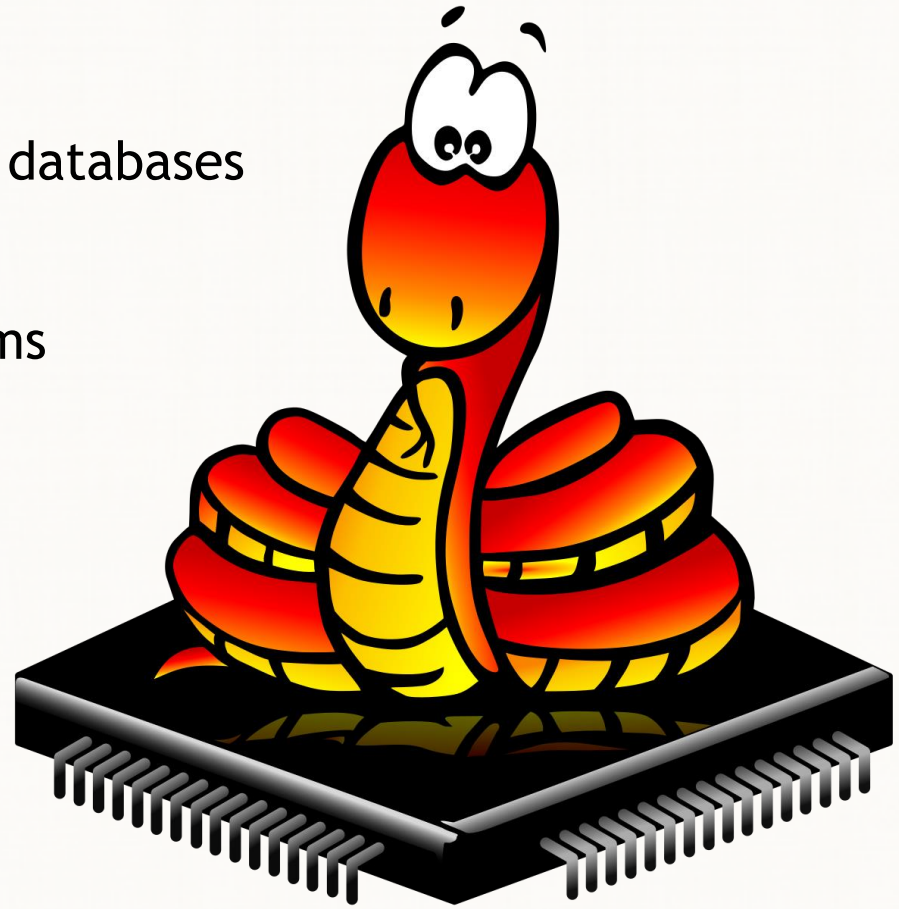
- **Multimedia Programming** – PyMedia, GStreamer, PyGame, Pyglet

Application Areas

- **Database Programming** – Supporting connectors for all databases
- **Networking** – Asyncio, Diesel, Pulsar, Twisted, NAPALM
- **Automation** – ANN, IOT, MicroPython, Embedded Systems

AsyncIO

MySQL®



Application Areas

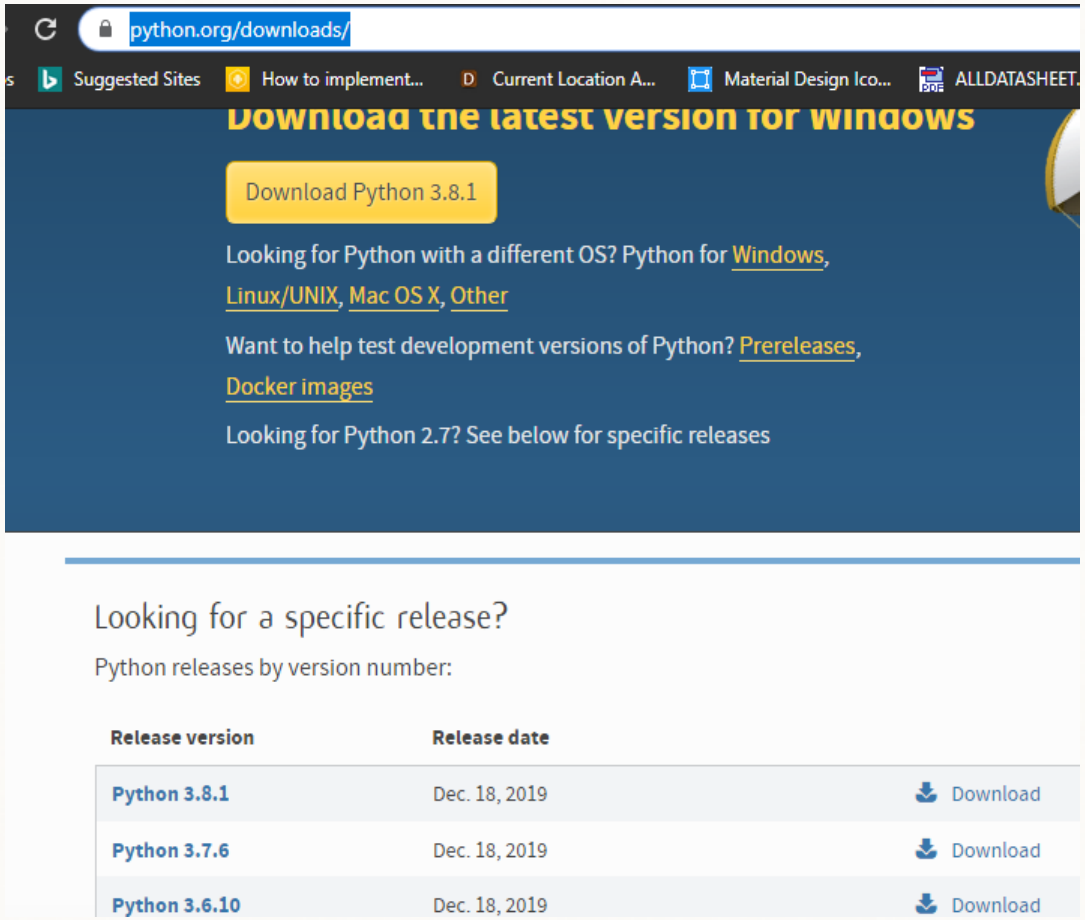
- **Web Scraping -** Beautiful Soup, Scrapy, Requests, Selenium
- **System Administration -** Fabric, Salt, Psutil, Ansible, Chef, Puppet, Blueprint, Buildout, Shinken
- **Scientific Computing -** Astrophy, Biopython, Bokeh, Cubes, Dask, Matplotlib, NetworkX, NumPy, Pandas, PyTorch, SciPy

Application Areas

- **Text Processing -** NLTK, Polyglot, TextBlob, CoreNLP, Pattern, Vocabulary, QuePy
- **Image Processing -** OpenCV, Pillow, Scikit-image
- **Machine Learning -** Scikit-learn, TensorFlow, Keras, Theano, PyTorch
- **Data Analytics -** NumPy, Pandas, Matplotlib, Seaborn, Plot.ly, SciPy
- **Data Science -** ML, DA, TensorFlow, Keras

How to get?

- Visit
  - https://www.python.org/downloads/

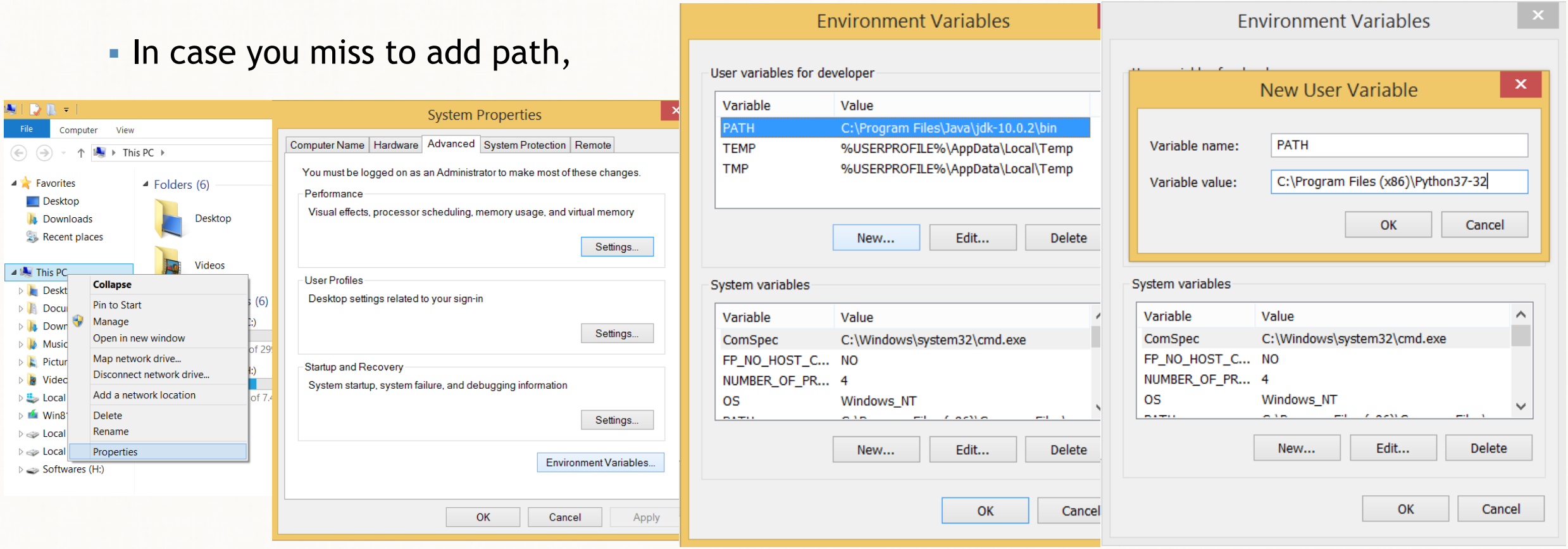


Install on Windows

- Run Installer
- Make sure you tick on “ADD PYTHON to PATH”

Setting Up Environment

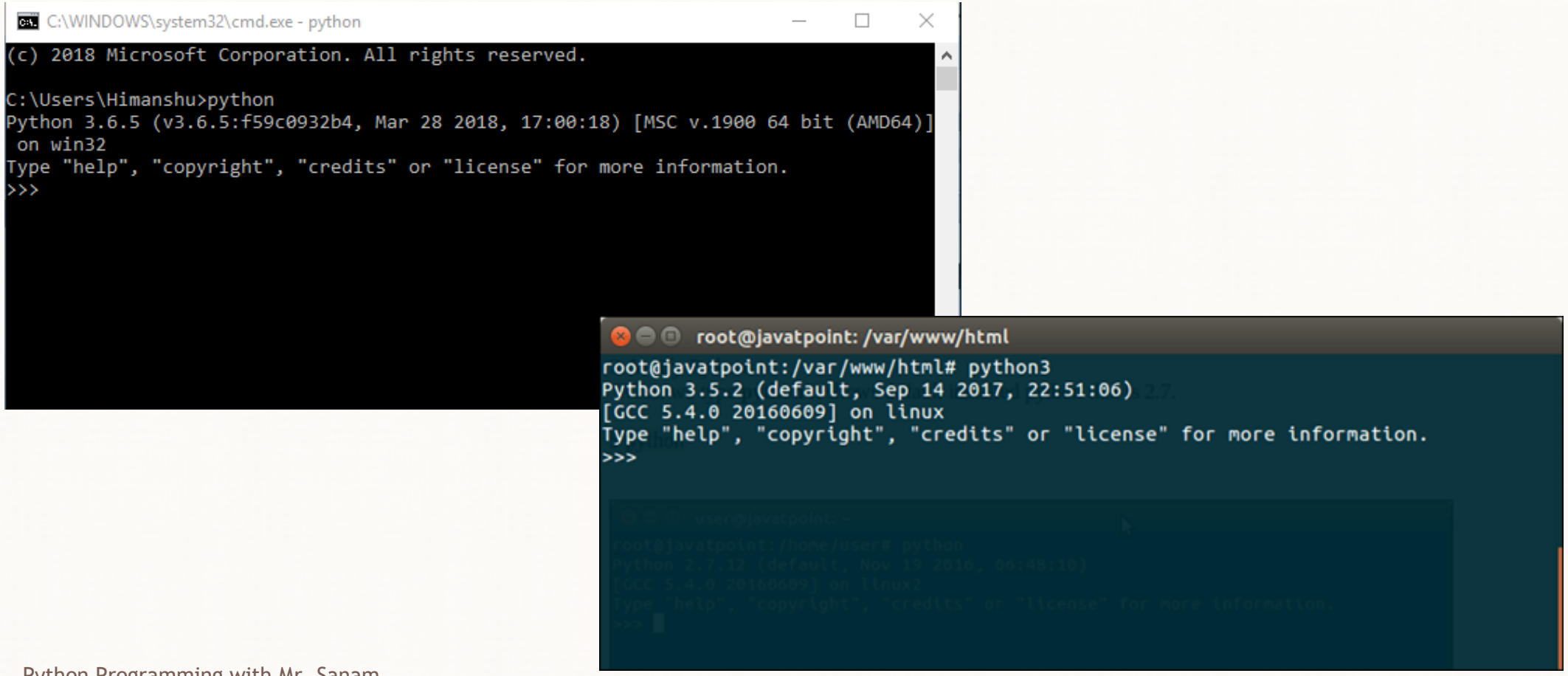
- In case you miss to add path,



Install on Linux (Ubuntu)

- sudo apt-get install python3
- sudo apt-get install idle3

Verify Installation

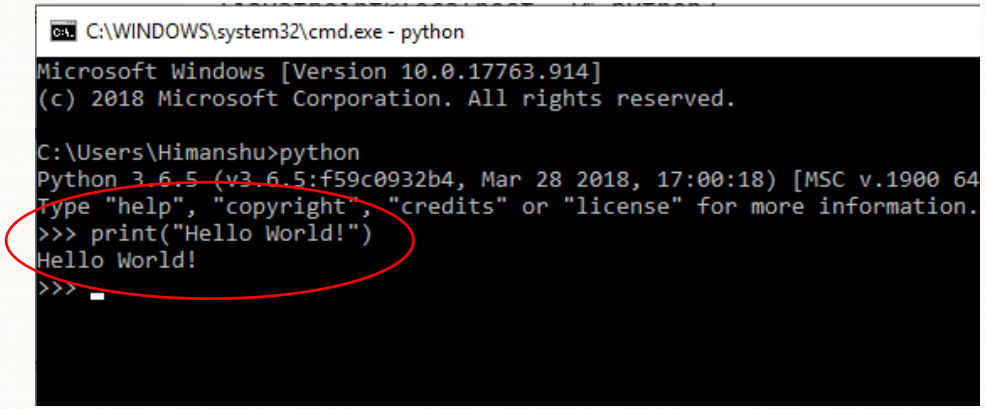


How to write Python Code?

- Python provides us the two ways to run a program:
  - Using Interactive interpreter prompt
  - Using a script file

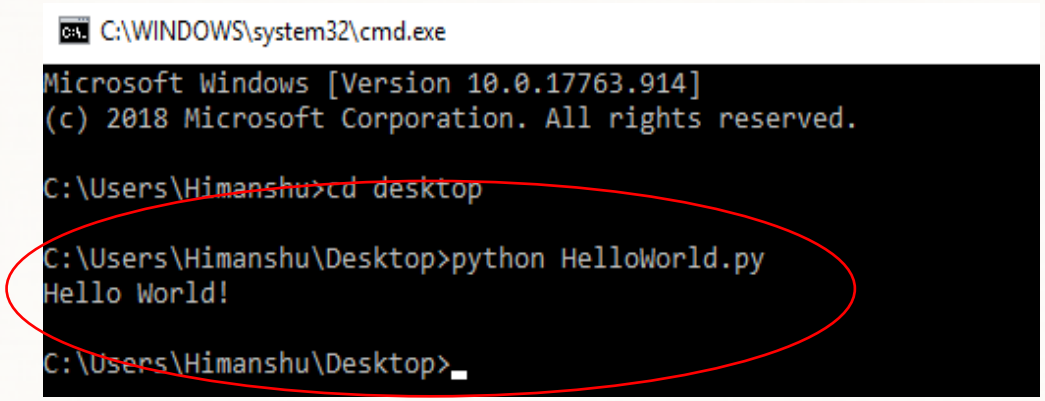
Using Interactive Interpreter Prompt

- Open Terminal/Command Prompt/Idle3
- In case of Terminal or Command Prompt
  - python or python3 (in case you have python2 and python3 both installed)
- Type
  - print("Hello World!")
  - and press ENTER key
- You will get a message in console showing
  - Hello World!



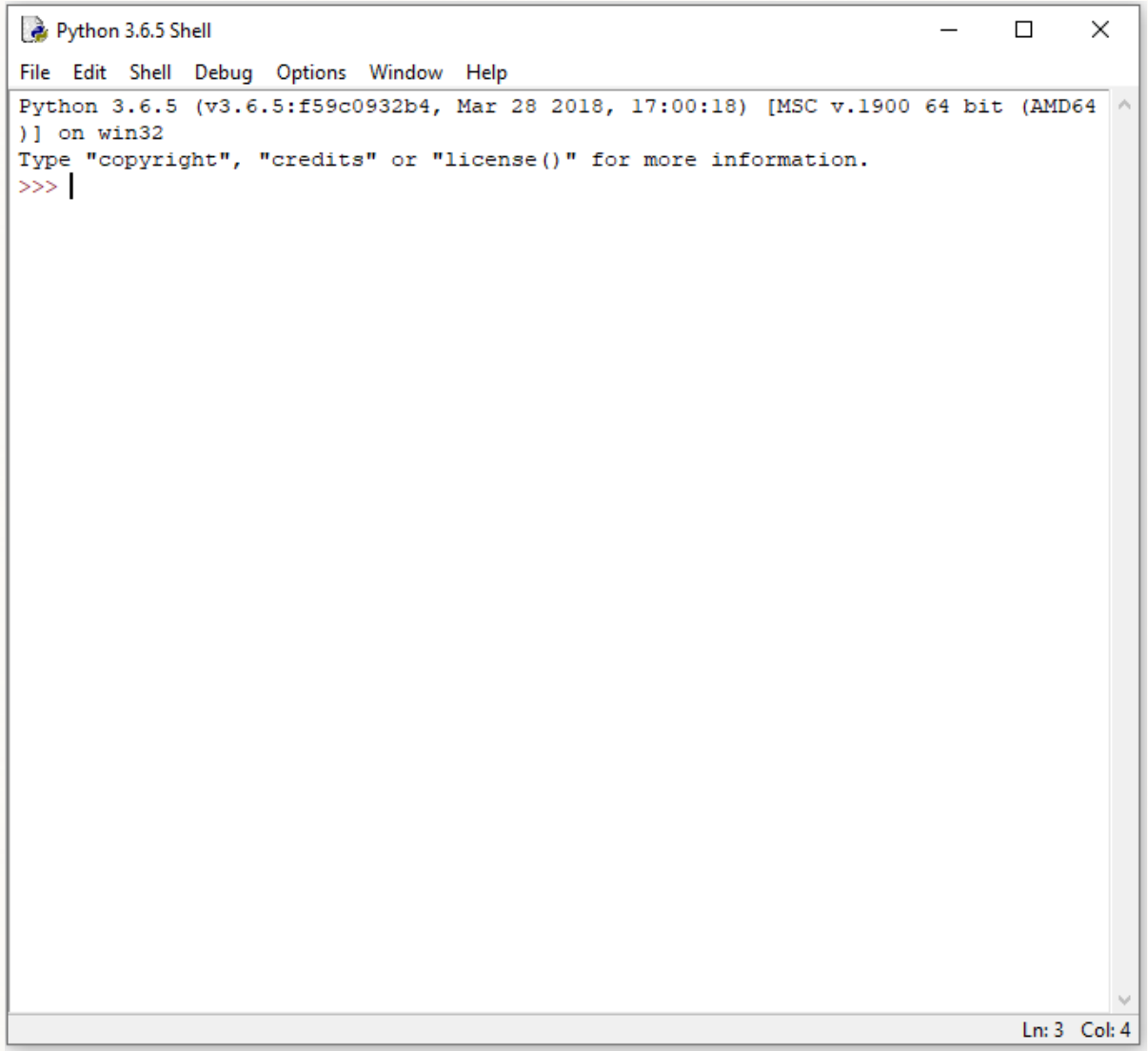
Using a script file

- Open your favorite code editor, type the following code
  - print(“Hello World!”)
- Save the file, lets say, HelloWorld.py (.py is python code extension, and file name can be anything with alpha-numerals)
- Now open a Terminal or Command Prompt in the very same folder of above file.
- And then type following command
  - python HelloWorld.py (or python3 HelloWorld.py)



But wait...

- Don’t get just bored with this tradition “Hello World” thing
- Lets find out, how python stand ahead of all
- Head on to your Python Terminal Window (or Idle Shell Window)



Now lets do something

- 4+5
- 8/3
- 4//2
- 19* “o”
- a = 34
- b = “good”
- print(b)
- print(a,b)

How Python Works?

- Python is an interpreted language i.e., Interpreted language requires **interpreter**, which takes the source code and **executes one instruction at a time.**
- **Now, what is an interpreter?**
- An interpreter is a program that is written in some other language and compiled into machine readable language. The interpreter itself is the machine language program and is written to read source programs from the interpreted language and interpret them.

Getting Output on Console

- The **print()** function
  - Syntax
    - print(*objects, sep=‘ ‘, end=‘\n’, file=sys.stdout, flush=False)
  - *objects – Objects to print (can be single or multiple).
  - sep – The separator string to separate two values in case of printing multiple objects.
  - end – String to be printed after all objects are printed.
  - file – The location where we want to print our output.
  - flush – Normally output to a file or the console is buffered, with text output at least until you print a newline. The flush makes sure that any output that is buffered goes to the destination.

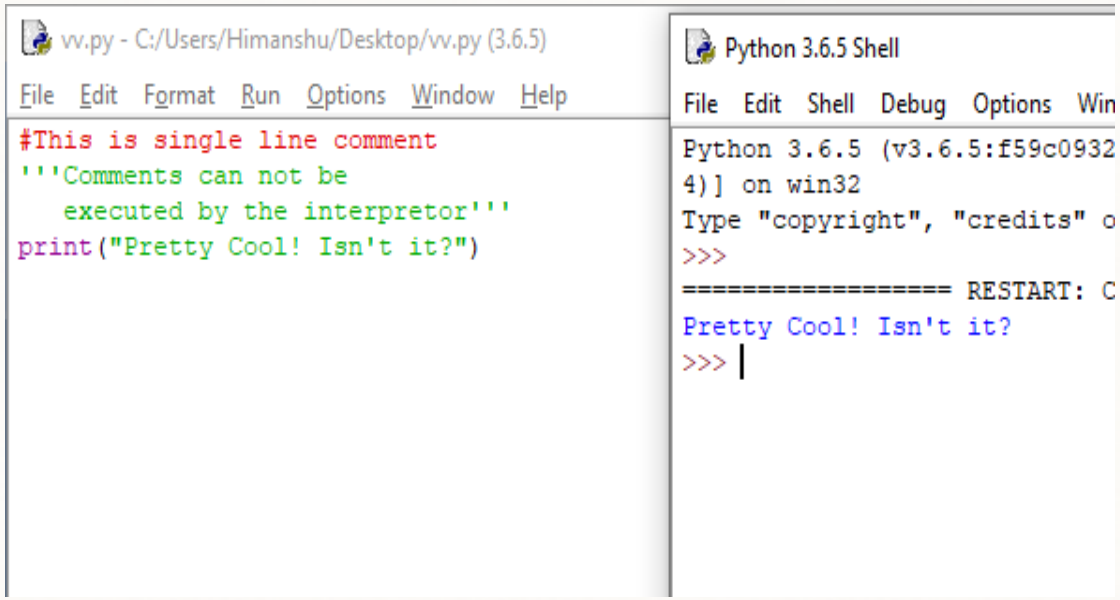
Python Comments

- Comments can be used to explain Python code.
- Comments can be used to make the code more readable.
- Comments can be used to prevent execution when testing code.
- Comments starts with a #, and Python will ignore them

```
#This is a comment
print("Hello, World!")
print("Hello, World!") #This is a comment
```

- We can use multi line comments also using triple quoted string (although it means something else, we will discuss it later on)

```
"""This is a comment
Multiline comment"""
```



Python Variables

- Variable is a name used to refer memory locations because we can't remember memory addresses and used to hold values.
- In Python, we don't need to specify type of the variable because Python is smart enough to get variable type.
- Variables can be named using combination of letters, digits and underscore (_) . No other special symbol can be used.
- Variables can begin with letter or underscore but not with digit.
- Python is case sensitive language.
- Variables can also be called as identifiers.
- Valid identifiers:- a123, _n, n_9
- Invalid identifiers:- 1a, n%4, n 9
- The equal (=) operator is used to assign value to a variable.
- Python doesn't bound us to declare variables before use. It allows us to create variable at required time.
- Example:

```
a = 10
name = "Python Developer"
```

Python Variables contd.

- Python support Multiple Assignments

```
x=y=z=50   #(Here, x,y,z have same value i.e., 50)
x,y=20,42  #(Here, x =20 and y=42)
```

- Examples to demonstrate the variables:-

```
a=10
b,c=20,30
x=y=z=50
print(a)
print(b)
print(c)
print(x)
print(y)
print(z)
```

Python Keywords

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class',
'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal',
'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with',
'yield']
>>>
```

Python Operators

- Arithmetic `+,-,*,/,//,**,%`
- Relational `>, <, ==, !=, >=, <=`
- Assignment `=, +=, -=, *=, /=, //=, **=, %=`
- Logical `and, or, not`
- Bitwise `&, |, ^, <<, >>`
- Identity `is, is not`
- Membership `in, not in`

Arithmetic Operators

Operator	Name	Example	Output
+	Addition	2+3	5
-	Subtraction	5-3	2
*	Multiplication	5*3	15
/	Division	5/2	2.5
//	Floor Division	5//2	2
%	Modulus	5%2	1
**	Exponentiation	5**2	25

Relational Operators

Operator	Example	Same As
=	X=5	X=5
+=	X+=3	X=X+3
-=	X-=3	X=X-3
*=	X*=3	X=X*3
/=	X/=3	X=X/3
%=	X%=3	X=X%3
//=	X//=3	X=X//3
**=	X**=3	X=X**3
&=	X&=3	X=X&3
|=	X|=3	X=X|3
^=	X^=3	X=X^3
>>=	X>>=3	X=X>>3
<<=	X<<=3	X=X<<3

Assignment Operators

Operator	Example	Same As
=	X=5	X=5
+=	X+=3	X=X+3
-=	X-=3	X=X-3
*=	X*=3	X=X*3
/=	X/=3	X=X/3
%=	X%=3	X=X%3
//=	X//=3	X=X//3
**=	X**=3	X=X**3
&=	X&=3	X=X&3
|=	X|=3	X=X|3
^=	X^=3	X=X^3
>>=	X>>=3	X=X>>3
<<=	X<<=3	X=X<<3

Logical Operators

Operator	Name	Example	Output
and	Logical And	2<5 and 2<10	True
or	Logical Or	2<5 or 2<1	True
not	Logical Not	not(2<5 and 2<10)	False

Bitwise Operators

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of the two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Left Shift	Shift bits to left pushing zeros in right
>>	Right Shift	Shift bits to right pushing zeros in left

Identity Operators

Operator	Description	Example
is	Returns True if both variables are same object	x is y
is not	Returns True if both variables are not same object	x in not y

Membership Operators

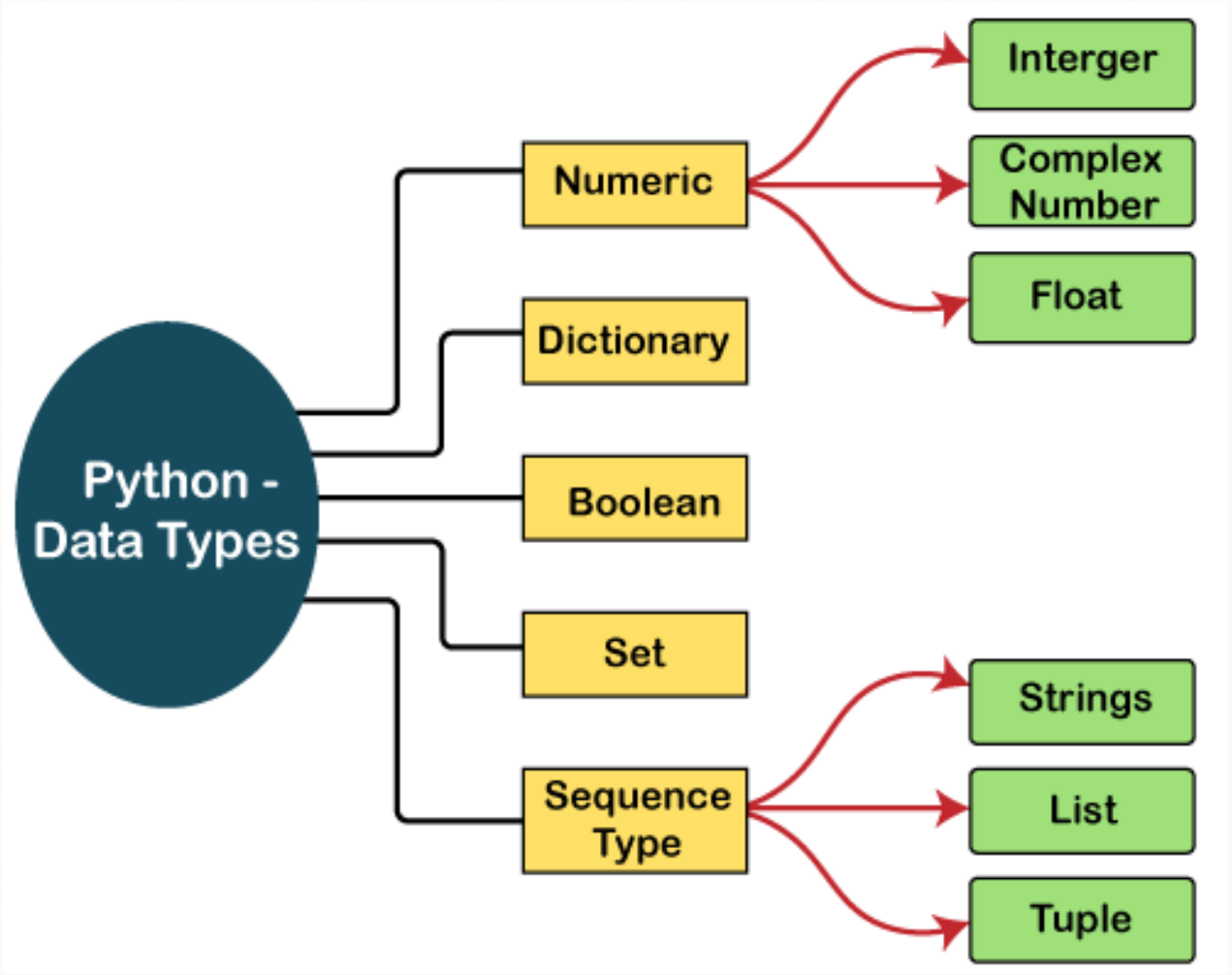
Operator	Description	Example
in	Returns True if a sequence have the specified value	x in y
not in	Returns True if a sequence do not have the specified value	x not in y

Use of Brackets in Python

- () Function Arguments, Expressions, Tuples
- {} Dictionary, Sets
- [] Lists, Indexing Slicing
- <> not used as brackets

Data Classes & Built-In Functions

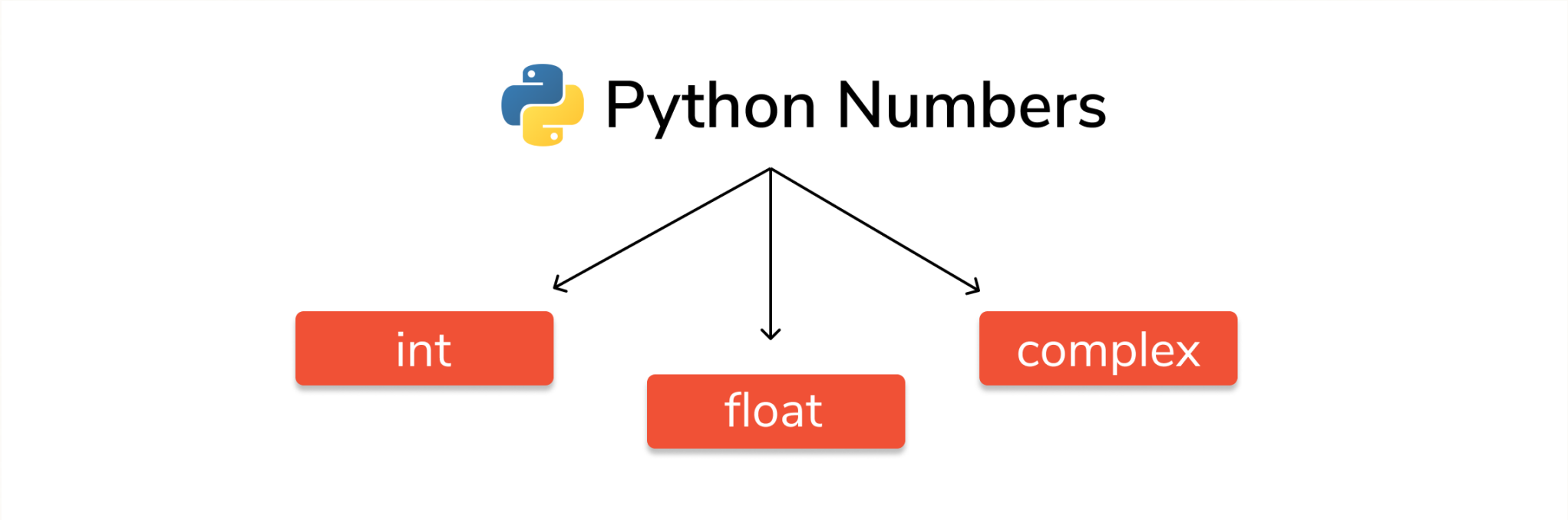
Data Types


Python -
Data Types
Numeric
Interger
Complex
Number
Float
Dictionary
Boolean
Set
Sequence
Type
Strings
List
Tuple

Name	Type	Description
Integers	int	Whole numbers, such as: **3 300 200**
Floating point	float	Numbers with a decimal point: **2.3 4.6 100.0**
Strings	str	Ordered sequence of characters: **"hello" 'Sammy' "2000" "楽しい"**
Lists	list	Ordered sequence of objects: **[10,"hello",200.3]**
Dictionaries	dict	Unordered Key:Value pairs: **{"mykey" : "value" , "name" : "Frankie"}**
Tuples	tup	Ordered immutable sequence of objects: **(10,"hello",200.3)**
Sets	set	Unordered collection of unique objects: **{"a","b"}**
Booleans	bool	Logical value indicating **True** or **False**

Numeric Category

Data Classes Built-in Functions


Python Numbers
int
float
complex

Int | Float | Complex

```
>>> a = 10
>>> print(a)
10
>>> type(a)
<class 'int'>
>>> print(type(a))
<class 'int'>
>>> a
10
>>> b = 10.3
>>> print(b)
10.3
>>> print(type(b))
<class 'float'>
>>> c = 3+4j
>>> print(c)
(3+4j)
>>> print(type(c))
<class 'complex'>
>>>
```

String

Data Classes Built-in Functions

string

- Strings are arrays of bytes representing Unicode characters.
- A String is a **sequence of characters**.
- A single character is simply a string with a length of 1.
- A string is always defined in between single or double quotes (‘ or “).
- Strings are **immutable**.
- To create multiline strings use triple quotes (’’’ or ”””)
- Strings can be accessed via three methods:
  - Indexing
  - Slicing
  - Loops

Python Strings contd.

- Example to create String variable :-

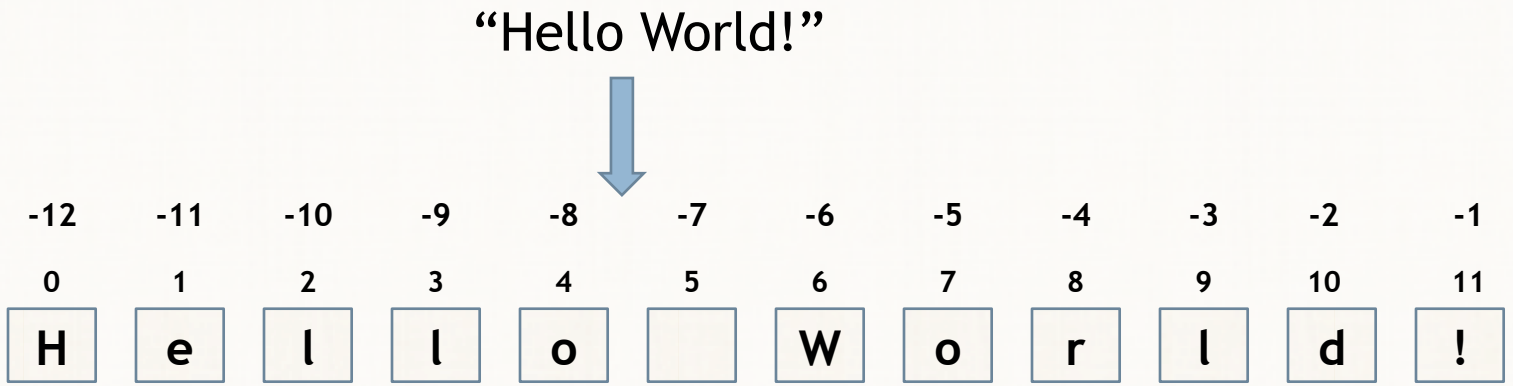
```
a="This is Single line string"
b="""This is Multliline
                string"""
print(type(a),type(b))
print(a, '\n', b)
```

```
a='This is single line string'
b="""This is multiline
string"""
print(type(a),type(b))
print(a,b,sep='\n')
```

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar
)] on win32
Type "copyright", "credits" or "lice
>>>
================== RESTART: C:\Users
<class 'str'> <class 'str'>
This is single line string
This is multiline
string
>>>
```

Accessing String using Indexing

- Index of strings starts with 0 and ends with (n-1). Here, n stands for the total number of characters in the string i.e., length of string.

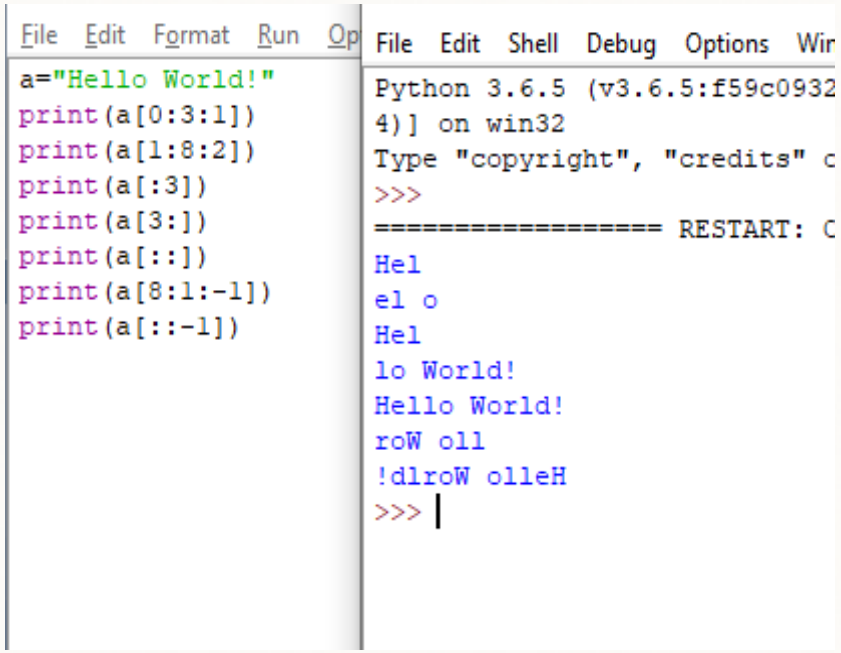


- To access string character by character use index number enclosed by square braces.
- Example:
  - `a="Hello World!"`
  - `print(a[0])`
  - `print(a[6])`

Accessing String using Slicing

- Syntax:
  - str[start:stop:step]
    - start – start index number (included)
    - stop – stop index number (excluded)
    - step – spacing between values and direction
- Example:

```
a="Hello World!"
print(a[0:3:1])
print(a[1:8:2])
print(a[:3])
print(a[3:])
print(a[::])
print(a[8:1:-1])
print(a[::-1])
```



Python String Operations

- Repeating the String

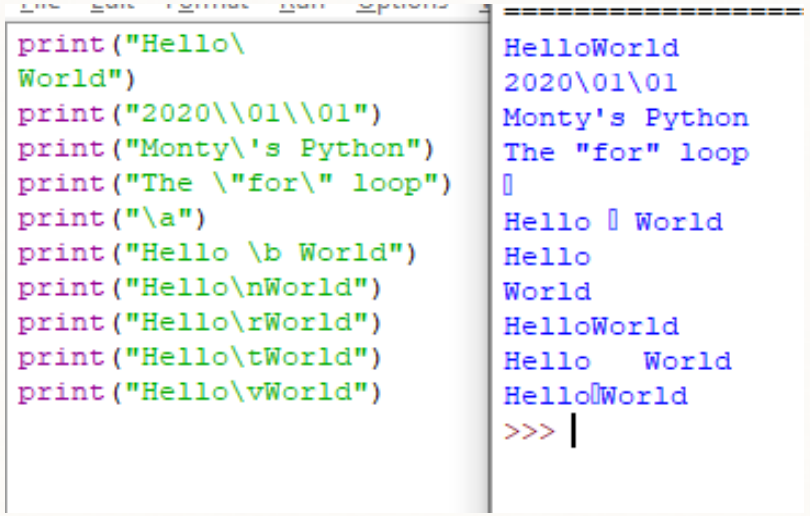
```
>>>
>>> a = 'good'
>>> a * 3
'goodgoodgood'
>>>
```

- Concatenation

```
>>>
>>> a = 'hello'
>>> b = 'bye'
>>> a + b
'hellobye'
>>>
```

String Escape Sequences

\<newline>		Ex:- print("Hello \ World")
\\	Backslash	Ex:- print("2020\\01\\01")
\'	Single quote	Ex:- print("Monty\'s Python")
\"	Double quote	Ex:- print("The \" for\" loop")
\a	ASCII Bell	Ex:- print("\a")
\b	ASCII Backspace	Ex:- print("Hello \b World")
\n	ASCII Linefeed	Ex:- print("Hello\nWorld")
\r	Carriage Return	Ex:- print("Hello\rWorld")
\t	Horizontal TAB	Ex:- print("Hello\tWorld")
\v	Vertical TAB	Ex:- print("Hello\vWorld")



Python String Functions

- capitalize()
  - Return a copy of the string with its first character capitalized and the rest lowercased.
- center(width[,fillchar])
  - Return centered in a string of length width. Padding is done using the specified fillchar.
- count(sub[,sub[,end]])
  - Return number of occurrences of substring sub in range[start,end].
- endswith(sub[,start[,end]])
  - Return True if string ends with specified substring sub in range[start,end].
- find(sub,[,start[,end]])
  - Return lowest index in the string where substring sub is found in range[start,end] else return -1.
- rfind(sub,[,start[,end]])
  - Return highest index in the string where substring sub is found in range[start,end] else return -1.
- index(sub,[,start[,end]])
  - Return lowest index in the string where substring sub is found in range[start,end] else raise ValueError.
- rindex(sub,[,start[,end]])
  - Return highest index in the string where substring sub is found in range[start,end] else raise ValueError.

Python String Functions

- lower()
  - Convert all upper case letters to lower case in the string.
- split(sep[,maxsplit])
  - Return a list of substrings of the string splitted on the basis of sep and length of maxsplit.
- join(words)
  - Concatenate a list or tuple of words separating with sep.
- lstrip()
  - Return a string with leading white spaces removed.
- rstrip()
  - Return a string with trailing white spaces removed.
- strip()
  - Return a string with leading and trailing both white spaces removed.
- swapcase()
  - Return copy of string after converting lowercase letters to uppercase letters and vice versa.
- upper()
  - Return copy of string after converting all characters to uppercase letters.
- replace(old, new)
  - Return a copy of string with all occurrences of substring old replaced by new.
- len(string)
  - Return length of the string i.e., number of characters in the string.

```
print("hello world".capitalize())
print("hello".center(50))
print("hello".center(50,'%'))
print("This is the example of counting".count('h'))
print("The Monty Python's Flying Circus".endswith('cus'))
print("The Monty Python's Flying Circus".find('y'))
print("The Monty Python's Flying Circus".rfind('y'))
print("The Monty Python's Flying Circus".lower())
print("The Monty Python's Flying Circus".upper())
print("The Monty Python's Flying Circus".swapcase())
print("Hello World".replace('World','Python'))
print("     Hello World     ".lstrip())
print("     Hello World     ".rstrip())
print("     Hello World     ".strip())
print("The Monty Python's Flying Circus".split("'"))
print('-'.join(['2020','01','01']))
```

```
Hello world
                         hello
%%%%%%%%%%%%%%%%%%%%%%%%hello%%%%%%%%%%%%%%%%%%%%%%%%
2
True
8
21
the monty python's flying circus
THE MONTY PYTHON'S FLYING CIRCUS
tHE mONTY pYTHON'S fLYING cIRCUS
Hello Python
Hello World
     Hello World
Hello World
['The Monty Python', 's Flying Circus']
2020-01-01
>>>
```

List

Data Classes Built-in Functions

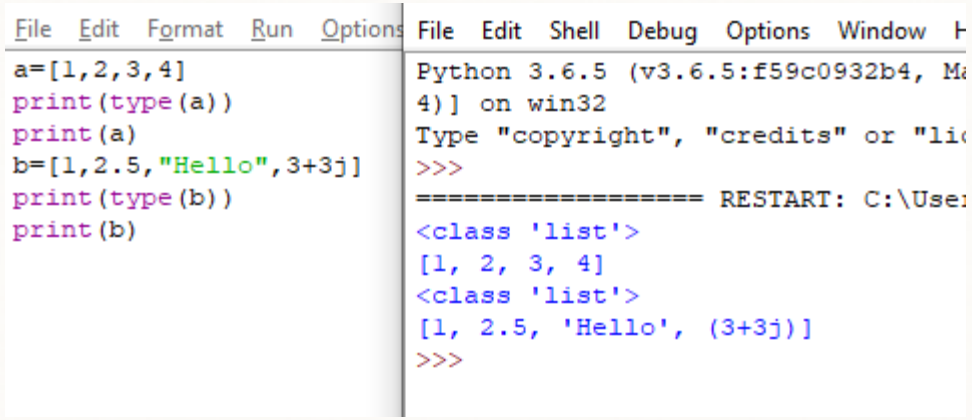
list

- Lists are just like the arrays.
- Lists may contain Homogenous or Heterogeneous data i.e., A list may contain integer data as well as float, string, list or any other type of data unlike arrays.
- Lists are **mutable** objects.
- A List is always defined in between squared braces [ ]
- Lists can be accessed via three methods:
  - Indexing
  - Slicing
  - Loops

Python Lists contd.

- Example to create a List object

```
a=[1,2,3,4]
print(type(a)
print(a)
b=[1,2.5,”Hello”,3+3j]
print(type(b))
print(b)
```



Accessing Lists using Indexing

- Index of lists starts with 0 and ends with (n-1). Here, n stands for the total number of elements in the list i.e., length of list.

[1,2.5,'Hello',3+3j]

0	1	2	3
1	2.5	'Hello'	3+3j

- To access list element by element use index number enclosed by square braces.
- Example
  - `a= [1,2.5,'Hello',3+3j]`
  - `print(a[0])`
  - `print(a[-2])`
  - `print(a[2][3])`

Accessing List using Slicing

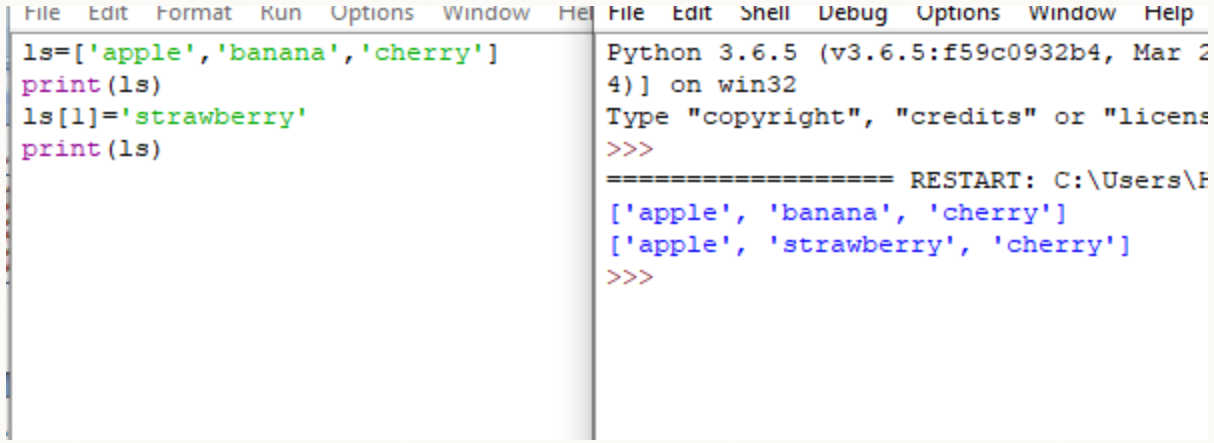
- Syntax :-
  - list[start:stop:step]
    - start – start index number (included)
    - stop – stop index number (excluded)
    - step – spacing between values and direction
- All rules of slicing we have seen in strings apply on lists also.
- Example

```
a= [1j, 2, 4, 4, “Python is easy!”,3, 3.5, 3, 6, 5]
print(a[0:3:1])
print(a[1:8:2])
print(a[:3])
print(a[-1:-4:-1])
```

Python List Operations

- Change Value of Item in a List
  - To change the value of specific item in a list, use index number and assign a value to it.
  - Example:

```
ls=[‘apple’, ‘banana’, ‘cherry’]
print(ls)
ls[1]=‘strawberry’
print(ls)
```



Python Lists Operations contd.

- Deleting a Item in List:

```
>>> a = ['apple', 'banana', 'cherry']
>>> del a[2]
>>> a
['apple', 'banana']
>>>
```

- Check if a item is in List:

```
>>> a = ['apple', 'banana', 'cherry']
>>> 'apple' in a
True
>>> 'mango' not in a
True
>>> 'papaya' in a
False
>>>
```

Python Lists Operations contd.

- Repeating Items of List

```
>>> a = ['apple', 'banana', 'cherry']
>>> a*2
['apple', 'banana', 'cherry', 'apple', 'banana', 'cherry']
>>>
```

- Concatenating Lists (Also used for insertion)

```
>>> a = ['apple']
>>> a += ['banana']
>>> a
['apple', 'banana']
>>> a += 'cherry'
>>> a
['apple', 'banana', 'c', 'h', 'e', 'r', 'r', 'y']
>>>
```

Python List Functions

- len(list)
  - Return length i.e., number of elements in a list.
- append(x)
  - Adds an item (x) to the end of the list.
- extend(iterable)
  - Extend the list by appending all the items from the iterable.
- insert(i,x)
  - Inserts an item (x) at a given index i.
- remove(x)
  - Removes the first item from the list that has a value of x. Returns an error if there is no such item.
- pop([i])
  - Remove and return the item at the given index in the list. If no index is specified, pop() removes and returns the last item in list.
- clear()
  - Removes all items from the list.
- index(x[,start[,end]])
  - Returns the position of the first list item that has a value of x. Raise a ValueError if there is no such item. The start and end arguments are used to limit the search for the item in a particular subsequence. Returned index is computed relative to the beginning of the full sequence.

Python List Functions

- count(x)
  - Returns the number of times x appears in the list.
- sort(key=None,reverse=False)
  - Sorts the items of the list in place. key specifies a function of one argument that is used to extract a comparison key from each list element. The default value is None (compares the elements directly). reverse Boolean value. If set to True, then the list elements are stored as if each comparison were reversed.
- reverse()
  - Reverses the elements of the list in place.
- copy()
  - Returns a shallow copy of the list.
- **PYTHON Built-in**
  - list([iterable])
    - list() constructor returns a mutable list of elements.
  - max(iterable,*,key,default)
    - Returns the largest item in inerrable or largest of two arguments.
  - min(iterable,*,key,default)
    - Returns the smallest item in inerrable or smallest of two arguments.

```
a = ["bee", "moth"]
print(len(a))
print(a)
a.append("ant")
print(a)
a.extend(["ant", "fly"])
print(a)
a.insert(1, "wasp")
print(a)
a.remove("moth")
print(a)
a.pop()
print(a)
a.pop(1)
print(a)
a.clear()
print(a)
a = ["bee", "ant", "moth", "ant"]
print(a.index("ant"))
print(a.index("ant", 2))
print(a.count("ant"))
print(a.count("wasp"))
a.sort()
print(a)
a.sort(reverse=True)
print(a)
a.sort(key=len)
print(a)
a.reverse()
print(a)
b=a
b.append("ant")
print(a)
print(b)
b = a.copy()
b.append("ant")
print(a)
print(b)
print(max(b))
print(min(b))
```

```
File  Edit  Shell  Debug  Options  Window  Help
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018,
4)] on win32
Type "copyright", "credits" or "license()" fo
>>>
==================
2
['bee', 'moth']
['bee', 'moth', 'ant']
['bee', 'moth', 'ant', 'ant', 'fly']
['bee', 'wasp', 'moth', 'ant', 'ant', 'fly']
['bee', 'wasp', 'ant', 'ant', 'fly']
['bee', 'wasp', 'ant', 'ant']
['bee', 'ant', 'ant']
[]
1
3
2
0
['ant', 'ant', 'bee', 'moth']
['moth', 'bee', 'ant', 'ant']
['bee', 'ant', 'ant', 'moth']
['moth', 'ant', 'ant', 'bee']
['moth', 'ant', 'ant', 'bee', 'ant']
['moth', 'ant', 'ant', 'bee', 'ant']
['moth', 'ant', 'ant', 'bee', 'ant']
['moth', 'ant', 'ant', 'bee', 'ant', 'ant']
moth
ant
>>>
```

Tuple

Data Classes Built-in Functions

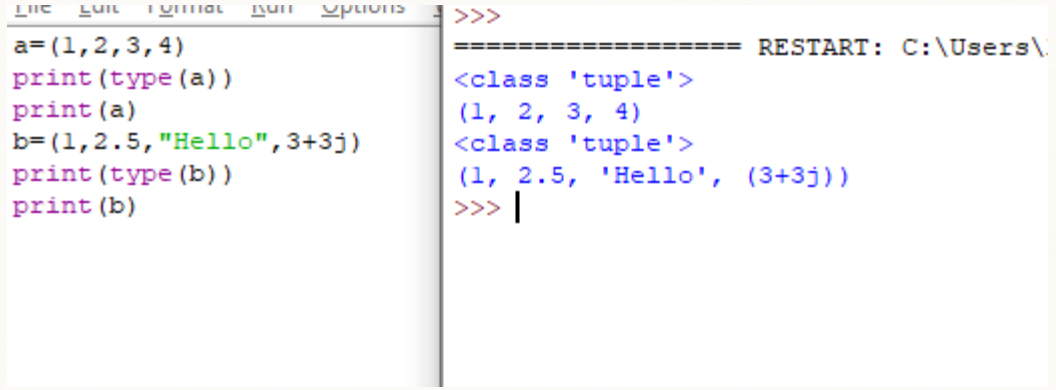
tuple

- Tuple is just like the arrays.
- Tuple may contain Homogenous or Heterogeneous data i.e., A Tuple may contain integer data as well as float, string, list or any other type of data unlike arrays.
- Tuple is immutable object.
- A tuple is always defined in between round braces ( )
- Tuple can also be accessed via three methods:
  - Indexing
  - Slicing
  - Loops

Python Tuple contd.

- Example to create a Tuple object

```
a=(1,2,3,4)
print(type(a))
print(a)
b=(1,2.5,“Hello”,3+3j)
print(type(b))
print(b)
```



Applying Functions on Tuple

- count(x)
  - Returns the number of times x appears in the tuple.
- index(x[,start[,end]])
  - Returns the position of the first tuple item that has a value of x. Raise a ValueError if there is no such item. The start and end arguments are used to limit the search for the item in a particular subsequence. Returned index is computed relative to the beginning of the full sequence.

- PYTHON Built-in
  - max(iterable,*,key,default)
    - Returns the largest item in inerrable or largest of two arguments.
  - min(iterable,*,key,default)
    - Returns the smallest item in inerrable or smallest of two arguments.
  - len(tuple)
    - Return length i.e., number of elements in a tuple.

Dictionary

Data Classes Built-in Functions

dict

- Dictionary is a collection which is unordered, mutable and indexed.
- Dictionary may contain Homogenous or Heterogeneous data i.e., A dictionary may contain integer data as well as float, string, list or any other type of data unlike arrays.
- A dictionary is always defined in between braces({key:value}).
- A dictionary contain key value pair relationships just like our ordinary oxford dictionary. A key can have multiple values in form of iterable associated with it.
- dictionary can be accessed via three methods:
  - Specifying key
  - Using functions (get(), values(), keys())
  - Loops

Python Dictionary contd.

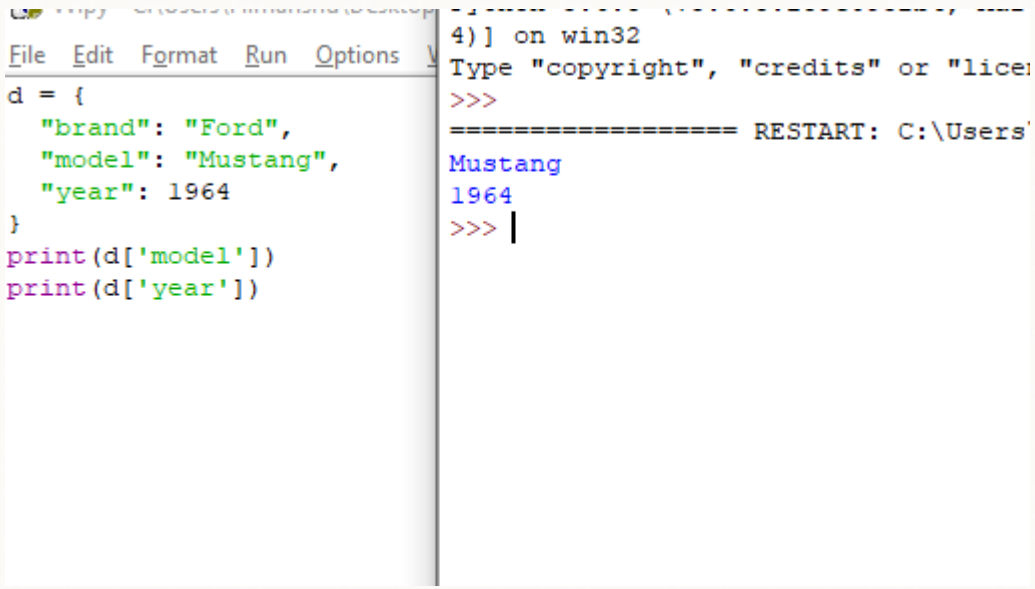
- Example to create a List object

```
d= {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(type(d))
print(d)
```

Accessing Dictionary Specifying Keys

- To access dictionary value use key enclosed by square braces.
- Example

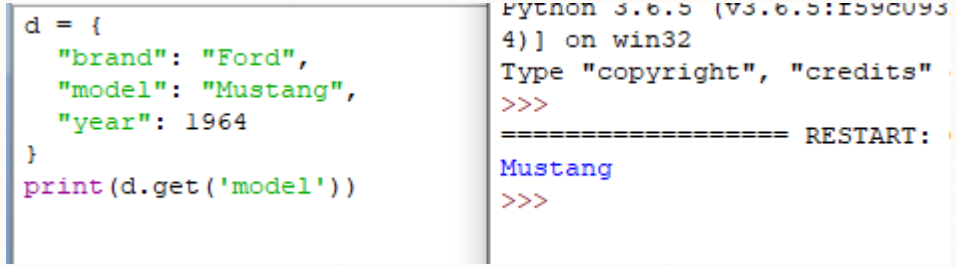
```
d= {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(d[‘brand’])
print(d[‘year’])
```



Accessing using Functions

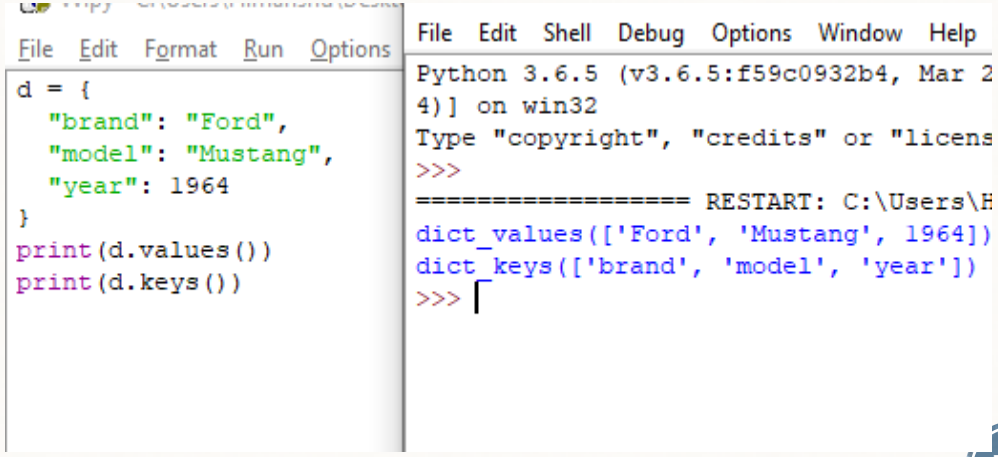
- get(key)
  - Used to access value of the key specified.

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

print(d.get('model'))
```



- values()
  - Return iterable object containing all values of dictionary.
- keys()
  - Return iterable object containing all keys of dictionary.

```
d= { "brand":  'Ford',"model": "Mustang","year": 1964}
print(d.values())
print(d.keys())
```



Python Dictionary Operations

- Change Value of Item in a dictionary
  - To change the value of specific item in a dictionary, use key and assign a value to it.
- Adding new key value pair in dictionary
  - Just create a new key and assign a value to it.

```
d= { "brand": "Ford“,
        "model": "Mustang",
        "year": 1964
}
print(d)
d[‘year’]=1985
print(d)
d[‘sales’]=1000
print(d)
```

```
d = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(d)
d['year']=1985
print(d)
d['sales']=1000
print(d)
```

Python 3.6.5 Shell

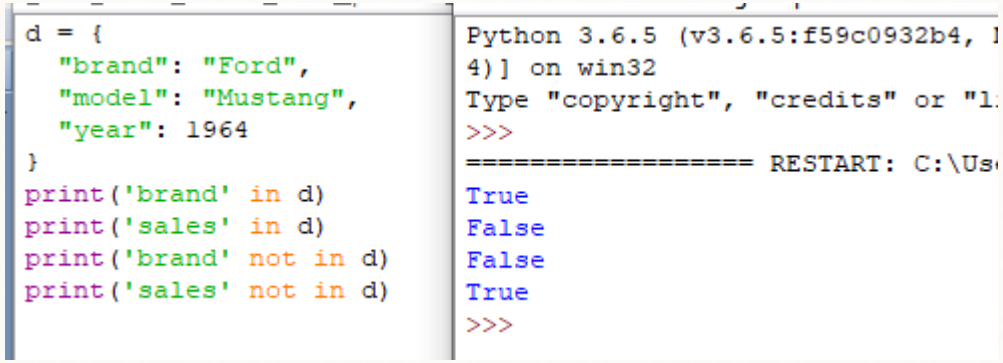
File Edit Shell Debug Options Window Help

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.190
4)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
==================                                          ======
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
{'brand': 'Ford', 'model': 'Mustang', 'year': 1985}
{'brand': 'Ford', 'model': 'Mustang', 'year': 1985, 'sales': 1000}
>>>
```

Python Dictionary Operations contd.

- Check Item if Exist in a dictionary:-To determine if a specified key is present in a dictionary or not, use the in and not in keyword.

```
d= {"brand": "Ford“,"model": "Mustang", "year": 1964}
print(‘brand’ in d)
print(‘sales‘ in d)
print(‘brand’ not in d)
print(‘sales‘ not in d)
```



Python Dictionary Operations contd.

- Merge two dictionaries using **kwargs

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
dict3 = {**dict1, **dict2}
print(dict3)
# Returns: {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

Python Dictionary Functions

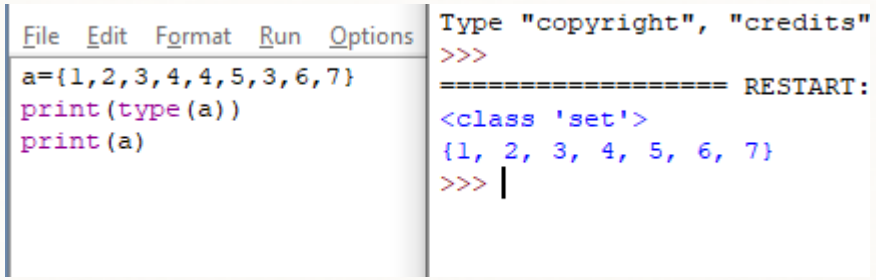
- len(dict)
  - Return length i.e., number of elements in a dictionary.
- pop(key)
  - Remove and return the value at the given key in the dictionary. If no key is specified, pop() removes and returns the value in dictionary.
- popitem ()
  - Remove last inserted item from dictionary.
- del(x)
  - Deletes the value or object specified by x.
- clear()
  - Removes all items from the dictionary.
- copy()
  - Returns a shallow copy of the dictionary.
- dict(key=value[,key=value…])
  - Return a dictionary made up of specified key value pairs.
- items()
  - Returns a list containing a tuple for each key value pair.

Set

Data Classes Built-in Functions

set

- A set is a collection which is unordered and unindexed. Tuple is mutable object.
- A set is always defined in between braces({ }).
- A set eliminates all repeated values and keeps a single copy of all elements.
- Set can be accessed via three methods:
  - Loops
  - Does not support indexing and slicing
- Example to create a Set object :-
  - `a={1,2,3,4,4,5,3,6,7}`
  - `print(type(a))`
  - `print(a)`



Python Set Functions

- add(x)
  - Adds value x in set.
- update([iterable])
  - Adds more than one items in set.
- remove(x)
  - Removes element x from set. If x not exists in set then it raises ValueError.
- clear()
  - It empties the set.
- set1.union(set2)
  - Return a new set with all items in both sets set1 and set2.
- set1.intersection(set2)
  - Return a set that contains the items common in set1 and set2.
- set1.intersection_update(set2)
  - Remove items that are not present in both set1 and set2 from set1.
- discard(x)
  - Removes element x from set. If x not exists in set then does not raises ValueError.
- pop()
  - Remove the last item from set and return it. Since sets are unordered So we cannot decide which element gets removed.

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
print(len(x))
x = {"apple", "banana", "cherry"}
print(max(x))
print(min(x))
x.add('berry')
print(x)
x.update(y)
print(x)
x.remove('google')
print(x)
x.clear()
print(x)
x = {"apple", "banana", "cherry"}
print(x.union(y))
print(x.intersection(y))
x.discard('apple')
print(x)
x.pop()
print(x)
del(x)
print(x)
```

```
Type "copyright", "credits" or "license()" for more information.
>>>
========== RESTART:                                          ========
3
cherry
apple
{'apple', 'cherry', 'berry', 'banana'}
{'microsoft', 'berry', 'banana', 'google', 'apple', 'cherry'}
{'microsoft', 'berry', 'banana', 'apple', 'cherry'}
set()
{'microsoft', 'banana', 'google', 'apple', 'cherry'}
{'apple'}
{'cherry', 'banana'}
{'banana'}
Traceback (most recent call last):
  File                                              in <module>
    print(x)
NameError: name 'x' is not defined
>>>
```

Creating Empty Collections

- None — Empty Variable
- ‘’ or “” — Empty String
- [ ] — Empty List
- ( ) — Empty Tuple
- { } — Empty Dictionary

Boolean

Data Classes Built-in Functions

bool

- Boolean is the pillar of decision making in Python Programming
- The keywords of Boolean data are **True**, **False**
- Anything non-zero or non-empty is termed as True in python
  - 1, [4, ‘good’], -2, ‘wow’, [()], “ ”
- And zero or empty is termed as False
  - ‘’, 0, []

Boolean operator	Boolean operation	Result
and	False and False	False
	True and True	True
	True and False	False
or	False or False	False
	True or True	True
	True or False	True
not	not True	False
	not False	True

Python Type-casting

- To specify one type of variable to other type we use type casting.
- For type casting we use constructor of the type class in which we want to cast the variable. This is due to the fact that Python is OOP, so it uses classes even for its primitive types.
  - int()
  - float()
  - str()
  - list()
  - dict()
  - tuple()
  - set()
  - bool()
  - frozenset()

Python Scripting

The concept of SUITES

Getting Input From Console

- The input() function :-
- Syntax :-

  input([prompt])

  - prompt – If specified, it is written to standard output without a trailing newline.
  - The function then reads a line from input.
  - Then converts it to string (stripping a trailing newline) and returns that.
  - When EOF is read, EOFError is raised.

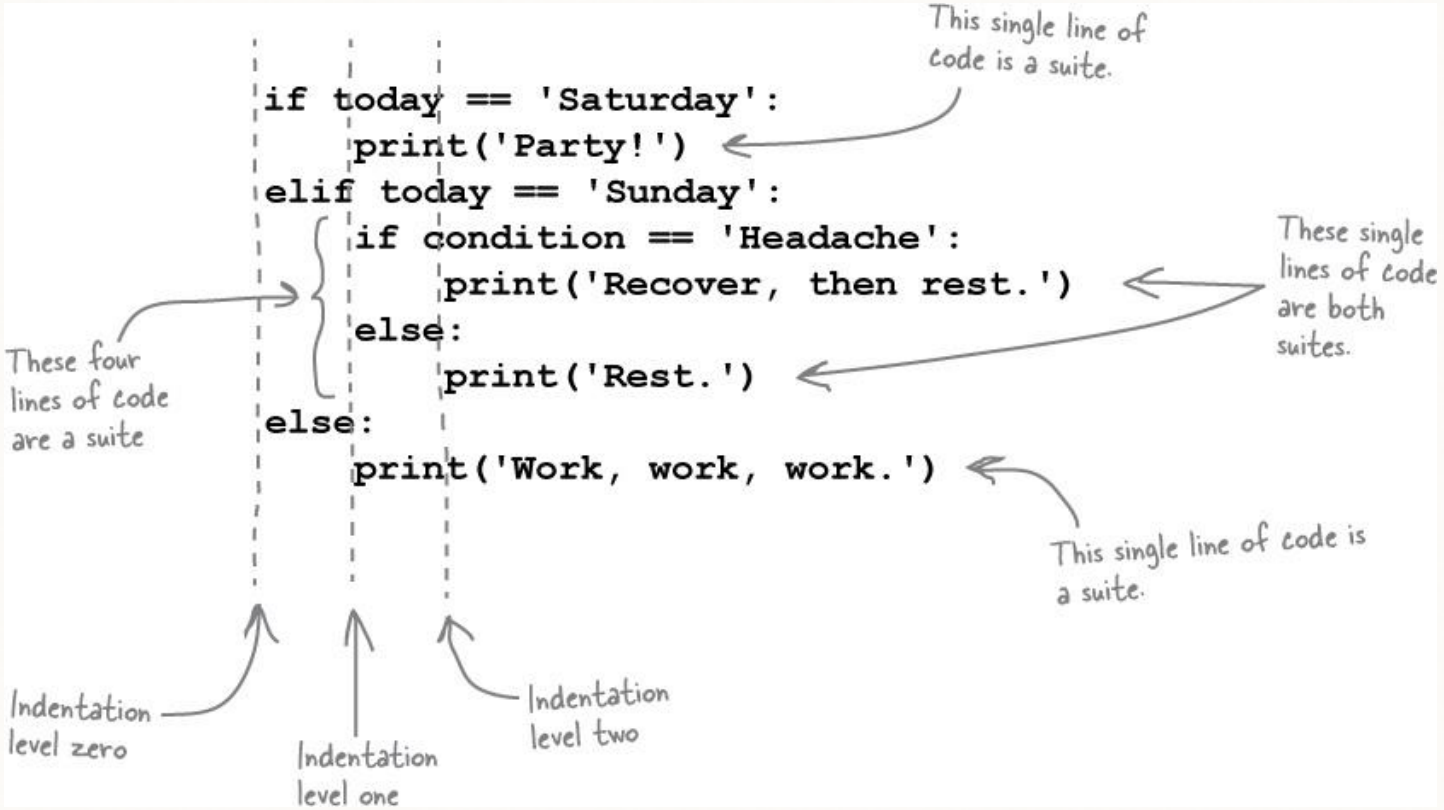
- Example:

```
a=input("Enter something in console")
print(type(a))
print(a)
```

```
Type "copyright", "credits" or "license()" f
>>>
========== RESTART: C:\Users\Himansh
Enter something in consolesomething
<class 'str'>
something
>>>
```

The Python SUITE

- The role of Indentation is very significant in Python

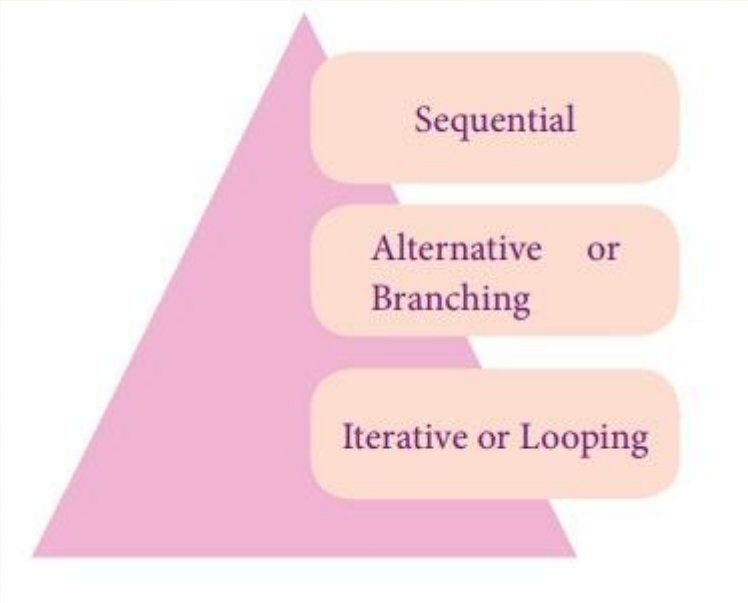


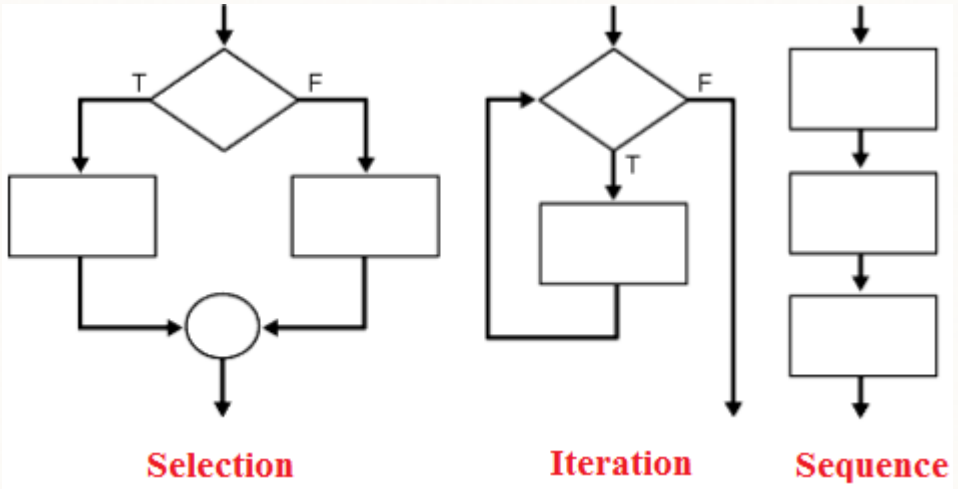
Control Statements

Python Programming

Types

- There are three important Control Structures in Python

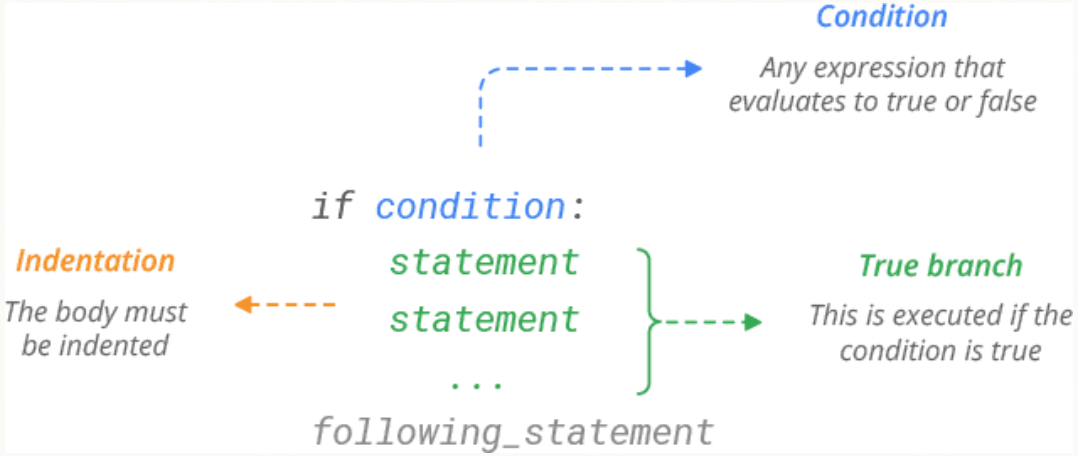




IF-ELIF-ELSE

Control Statements

if statement FLOW



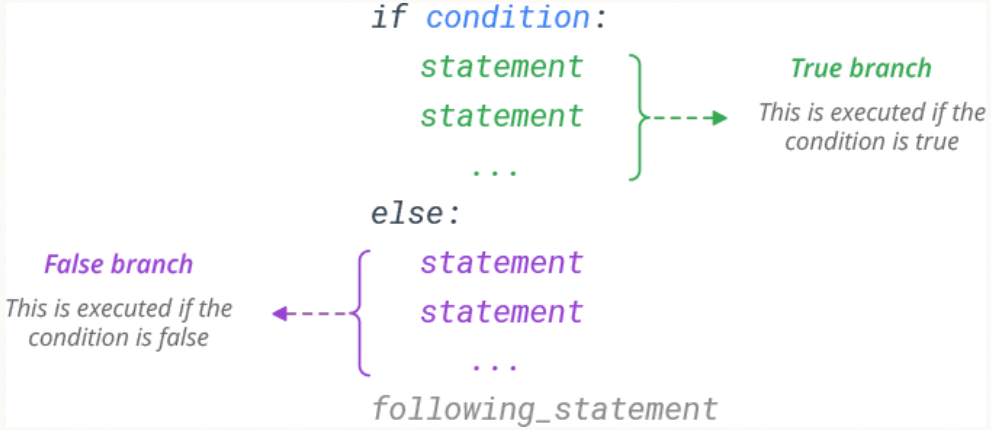

```
x, y = 7, 5
if not x < y:
    print('x is greater')

# Prints x is greater
```

```
x, y = 7, 5
if x > y:
    print('x is greater')

# Prints x is greater
```

```
x, y, z = 7, 4, 2
if x > y and x > z:
    print('x is greater')

# Prints x is greater
```

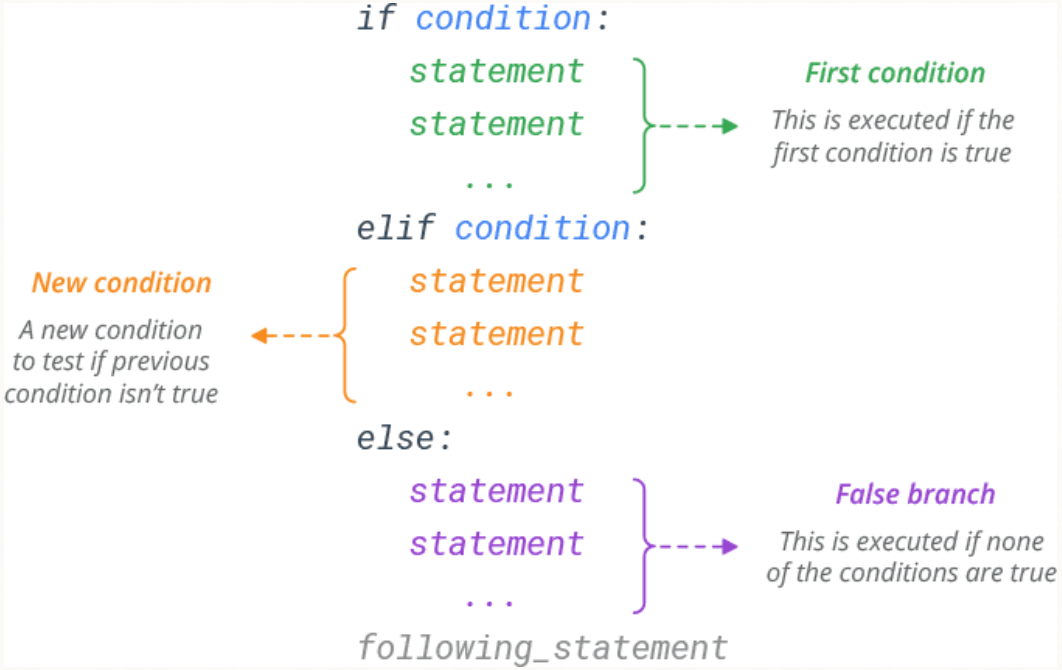
```
x, y, z = 7, 4, 9
if x > y or x > z:
    print('x is greater than y or z')

# Prints x is greater than y or z
```

if-else FLOW



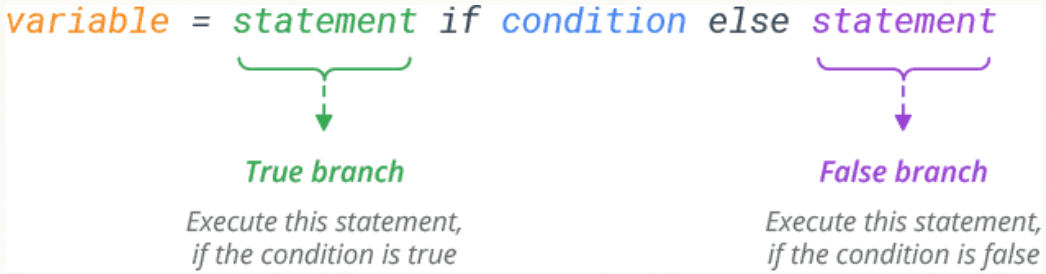

```
x, y = 7, 5
if x < y:
    print('y is greater')
else:
    print('x is greater')

# Prints x is greater
```

if-elif-else FLOW




```
x, y = 5, 5
if x > y:
    print('x is greater')
elif x < y:
    print('y is greater')
else:
    print('x and y are equal')

# Prints x and y are equal
```

Conditional Expression (Ternary Operator)




```
x, y = 7, 5
print('x is greater') if x > y else print('y is greater')

# Prints x is greater
```

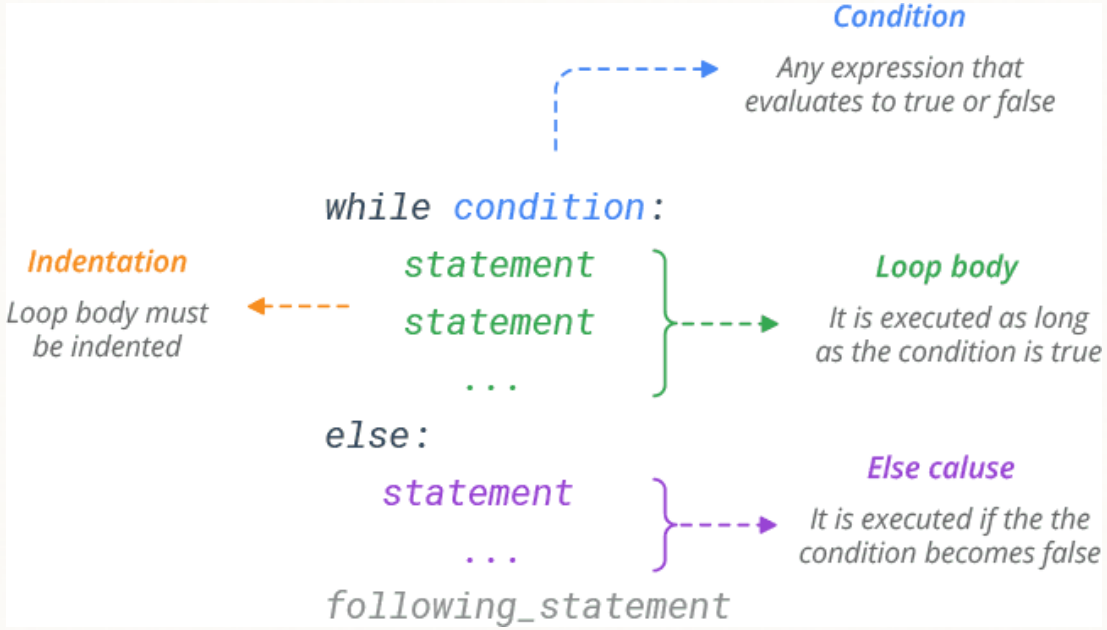
```
x, y = 7, 5
max = x if x > y else y
print(max)
# Prints 7
```

WHILE

Loop Statements

while

- A while loop is used when you want to perform a task indefinitely, until a particular condition is met. It's a condition-controlled loop.




```
# Iterate until string is empty
x = 'blue'
while x:
    print(x)
    x = x[1:]
# Prints blue
# Prints lue
# Prints ue
# Prints e
```

While loop examples

```
# Iterate until x becomes 0
x = 6
while x:
    print(x)
    x -= 1
# Prints 6 5 4 3 2 1
```

```
# Iterate until list is empty
L = ['red', 'green', 'blue']
while L:
    print(L.pop())
# Prints blue green red
```

```
# Iterate until string is empty
x = 'blue'
while x:
    print(x)
    x = x[1:]
# Prints blue
# Prints lue
# Prints ue
# Prints e
```

```
# Exit condition is false at the start
x = 0
while x:
    print(x)
    x -= 1
```

Break and Continue

- Python break statement is used to exit the loop immediately. It simply jumps out of the loop altogether, and the program continues after the loop.

```
# Exit when x becomes 3
x = 6
while x:
    print(x)
    x -= 1
    if x == 3:
        break
# Prints 6 5 4
```

- The continue statement skips the current iteration of a loop and continues with the next iteration.

```
# Skip odd numbers
x = 6
while x:
    x -= 1
    if x % 2 != 0:
        continue
    print(x)
# Prints 4 2 0
```

Else in While Loop

```
x = 6
while x:
    print(x)
    x -= 1
else:
    print('Done!')
# Prints 6 5 4 3 2 1
# Prints Done!
```

```
x = 0
while x:
    print(x)
    x -= 1
else:
    print('Done!')
# Prints Done!
```

- If the loop terminates prematurely with break, the else clause won't be executed.

```
x = 6
while x:
    print(x)
    x -= 1
    if x == 3:
        break
else:
    print('Done!')
# Prints 6 5 4
```

Infinite Loop

```
# Infinte loop with while statement
while True:
    print('Press Ctrl+C to stop me!')
```

```
# Loop runs until the user enters 'stop'
while True:
    name = input('Enter name:')
    if name == 'stop': break
    print('Hello', name)

# Output:
# Enter name:Bob
# Hello Bob
# Enter name:Sam
# Hello Sam
# Enter name:stop
```

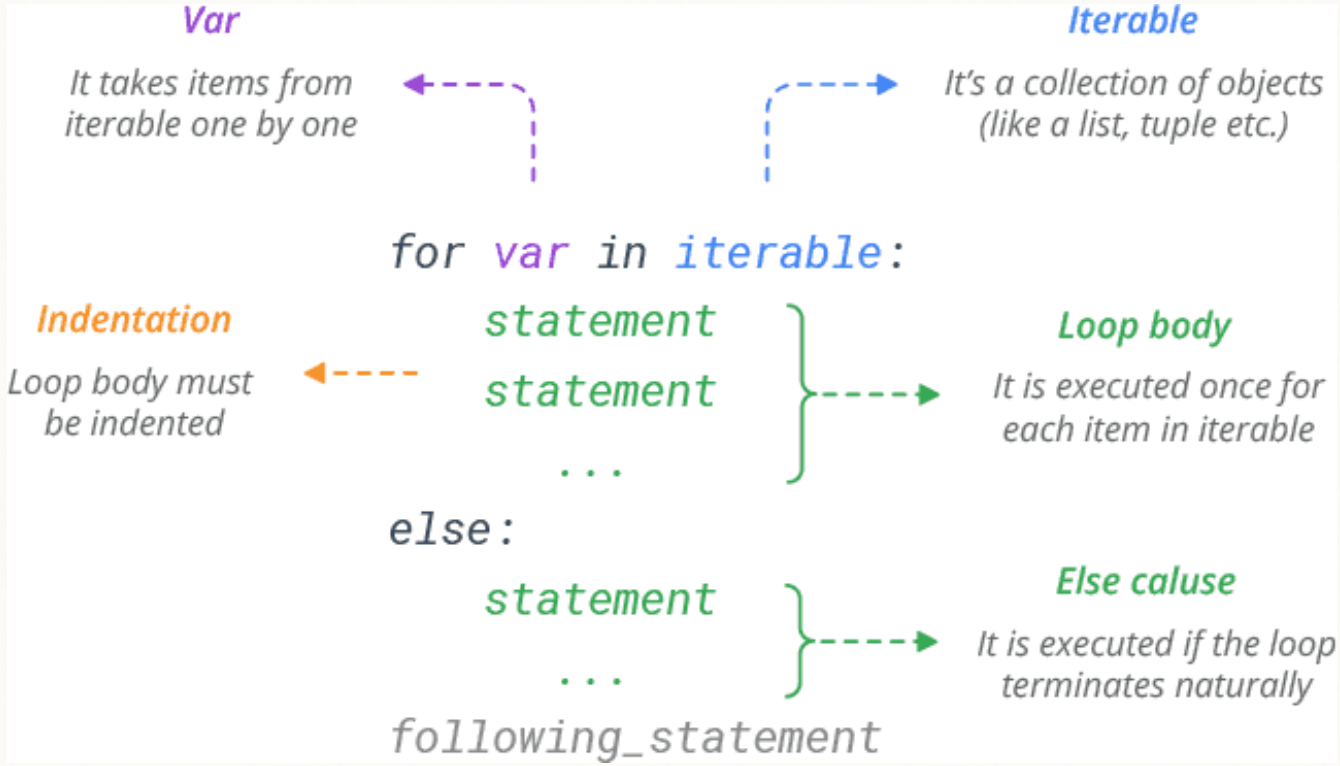
FOR

Loop Statements

FOR Loop

- The for statement in Python is a bit different from what you usually use in other programming languages.
- Rather than iterating over a numeric progression, Python's ***for*** statement iterates over the items of any *iterable* (list, tuple, dictionary, set, or string).
- The items are iterated in the order that they appear in the *iterable*.

For Loop contd.




```
# Iterate through a list
colors = ['red', 'green', 'blue', 'yellow']
for x in colors:
    print(x)
# Prints red green blue yellow
```

```
# Iterate through a string
S = 'python'
for x in S:
    print(x)
# Prints p y t h o n
```

```
# Flatten a nested list
list = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
for sublist in list:
    for number in sublist:
        print(number)
# Prints 1 2 3 4 5 6 7 8 9
```

Break and Continue in For Loop

```
# Break the loop at 'blue'
colors = ['red', 'green', 'blue', 'yellow']
for x in colors:
    if x == 'blue':
        break
    print(x)
# Prints red green
```

```
# Skip 'blue'
colors = ['red', 'green', 'blue', 'yellow']
for x in colors:
    if x == 'blue':
        continue
    print(x)
# Prints red green yellow
```

Else in FOR Loop

```
colors = ['red', 'green', 'blue', 'yellow']
for x in colors:
    print(x)
else:
    print('Done!')
# Prints red green blue yellow
# Prints Done!
```

```
colors = ['red', 'green', 'blue', 'yellow']
for x in colors:
    if x == 'blue':
        break
    print(x)
else:
    print('Done!')
# Prints red green
```

Using range( ) function

- The range(start,stop,step) function generates a sequence of numbers from 0 up to (but not including) specified number.

```
# Print 'Hello!' three times
for x in range(3):
    print('Hello!')
# Prints Hello!
# Prints Hello!
# Prints Hello!
```

```
# Generate a sequence of numbers from 2 to 6
for x in range(2, 7):
    print(x)
# Prints 2 3 4 5 6
```

```
for x in range(-5,0):
    print(x)
# Prints -5 -4 -3 -2 -1
```

```
# Increment the range with 2
for x in range(2, 7, 2):
    print(x)
# Prints 2 4 6
```

FOR Loop Operations

- Lopping through multiple Lists

```
# Loop through two lists at once
name = ['Bob', 'Sam', 'Max']
age = [25, 35, 30]
for x, y in zip(name, age):
    print(x, y)
# Prints Bob 25
# Prints Sam 35
# Prints Max 30
```

- Access Index in Loop

```
colors = ['red', 'green', 'blue']
for index in range(len(colors)):
    print(index, colors[index])
# Prints 0 red
# Prints 1 green
# Prints 2 blue
```

```
colors = ['red', 'green', 'blue']
for index, value in enumerate(colors):
    print(index, value)
# Prints 0 red
# Prints 1 green
# Prints 2 blue
```

Comprehension

Loop Statements

Comprehensions

- The Comprehensions are one-liner elegant approach to create the respective object
- In Python we have four comprehensions available:
  - List [output for-loop]
  - Dictionary { key:value for-loop }
  - Set { output for-loop }
  - Generator (output for-loop)
- The basic syntax of any comprehension does have two forms
  - `Bracket_open <output> for-loop Bracket_close`
  - `Bracket_open <output> for-loop if-condition Bracket_close`

List Comprehension

```
[expression for var in iterable]
```

Expression	*Var*	*Iterable*
It is evaluated once for each item in iterable	*It takes items from an iterable one by one*	*It's a collection of objects (like a list, tuple etc.)*

```
L = [x*3 for x in 'RED']
print(L)
# Prints ['RRR', 'EEE', 'DDD']
```

```
# Convert list items to absolute values
vec = [-4, -2, 0, 2, 4]
L = [abs(x) for x in vec]
print(L)
# Prints [4, 2, 0, 2, 4]
```

```
L = [(x, x**2) for x in range(4)]
print(L)
# Prints [(0, 0), (1, 1), (2, 4), (3, 9)]
```

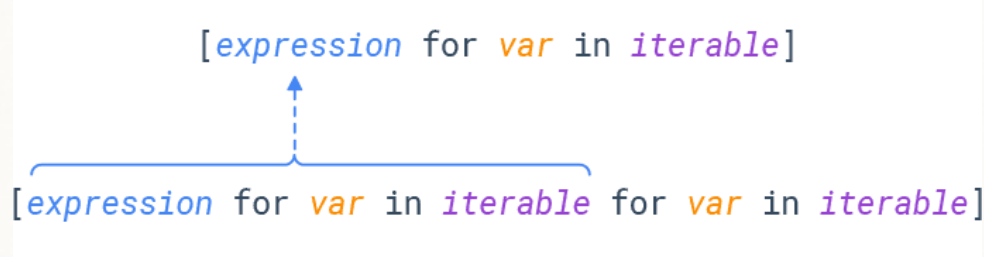
List Comprehensions with IF

```
[expression for var in iterable if_clause]
```

```
# Filter list to exclude negative numbers
vec = [-4, -2, 0, 2, 4]
L = [x for x in vec if x >= 0]
print(L)
# Prints [0, 2, 4]
```

```
vec = [-4, -2, 0, 2, 4]
L = []
for x in vec:
    if x >= 0:
        L.append(x)
print(L)
# Prints [0, 2, 4]
```

Nested List Comprehension




```
# With list comprehension
vector = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
L = [number for list in vector for number in list]
print(L)
# Prints [1, 2, 3, 4, 5, 6, 7, 8, 9]

# equivalent to the following plain, old nested loop:
vector = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
L = []
for list in vector:
    for number in list:
        L.append(number)
print(L)
# Prints [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Dictionary Comprehension

```
D = {}
for x in range(5):
    D[x] = x**2

print(D)
# Prints {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

```
D = {x: x**2 for x in range(5)}
print(D)
# Prints {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

```
{key:value for var in iterable}
```

key: value	*var*	*Iterable*
Key & value can be any expression & evaluated once for each item in iterable	*It takes items from an iterable one by one*	*It's a collection of objects (like a list, tuple etc.)*

```
D = {0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F'}
removeKeys = [0, 2, 5]

X = {k: D[k] for k in D.keys() - removeKeys}

print(X)
# Prints {1: 'B', 3: 'D', 4: 'E'}
```

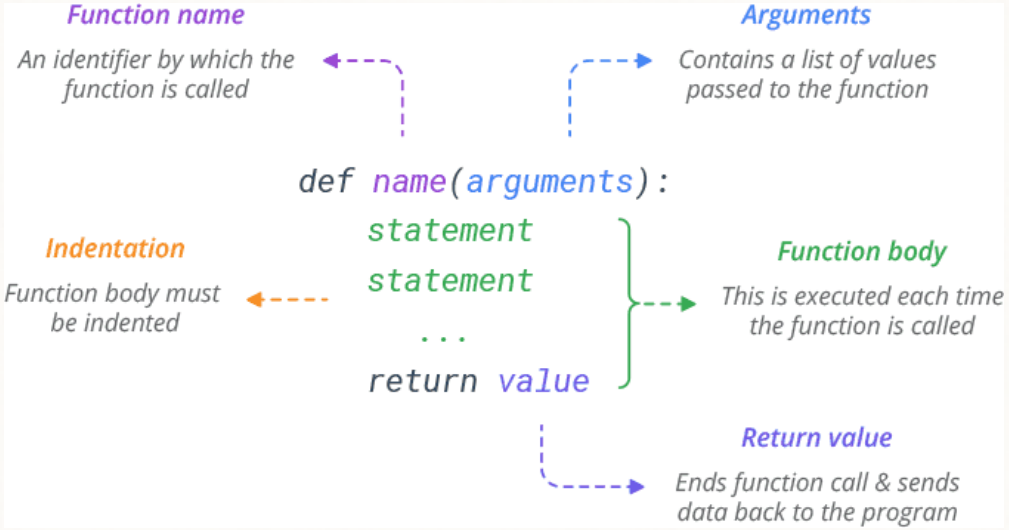
User Defined Functions

Python Functions

Python User Defined Functions

- A block of code which only runs when it is called.
- Enables code reusability.
- We can pass data, known as parameters.
- A function can return data as a result.
- We use def keyword to create a function.
- A block is created, So pay attention to Indentations.
- Build to perform a specific task.
- Types of Functions
  - Parameterized and Non Parameterized functions
  - Anonymous Functions/Lambda Expressions
  - Generator Functions
  - Decorator Functions

UDFs



- Creating

```
def hello():
    print('Hello, World!')
```

- Calling

```
hello()
# Prints Hello, World!
```

Non-parameterized UDFs

- Does not accept any parameters or arguments.
- Simply called using their name to perform the specific task.
- Syntax

```
def <FunctionName>():                #Block is created, Need Indentations
        Body                         #body of the functions to execute
FunctionName()                       #calling the function
```

```
def hello():
    print('Hello, World!')

hello()
# Prints Hello, World!
```

Parameterized UDFs

- Accept any number of parameters or arguments.
- Simply called using their name and passing required parameters in function.
- Syntax

```
def <FunctionName>(para1[,paraN]):
        Body
FunctionName(val1[,valN])
```

```
# Pass two arguments
def func(name, job):
    print(name, 'is a', job)

func('Bob', 'developer')
# Prints Bob is a developer
```

Types of Arguments

- Positional Arguments
- Keyword Arguments
- Default Arguments
- Variable Length Positional Arguments (*args)
- Variable Length Keyword Arguments (**kwargs)

Positional Arguments

```
# Pass two arguments
def func(name, job):
    print(name, 'is a', job)

func('Bob', 'developer')
# Prints Bob is a developer
```

Keyword Arguments

```
# Keyword arguments can be put in any order
def func(name, job):
    print(name, 'is a', job)

func(name='Bob', job='developer')
# Prints Bob is a developer

func(job='developer', name='Bob')
# Prints Bob is a developer
```

Default Arguments

```
# Set default value 'developer' to a 'job' parameter
def func(name, job='developer'):
    print(name, 'is a', job)

func('Bob', 'manager')
# Prints Bob is a manager

func('Bob')
# Prints Bob is a developer
```

Variable Length Arguments (*args and **kwargs)

- When you prefix a parameter with an asterisk * , it collects all the unmatched positional arguments into a tuple.
- Because it is a normal tuple object, you can perform any operation that a tuple supports, like indexing, iteration etc.

```
def print_arguments(*args):
    print(args)

print_arguments(1, 54, 60, 8, 98, 12)
# Prints (1, 54, 60, 8, 98, 12)
```

- The ** syntax is similar, but it only works for keyword arguments.
- It collects them into a new dictionary, where the argument names are the keys, and their values are the corresponding dictionary values.

```
def print_arguments(**kwargs):
    print(kwargs)

print_arguments(name='Bob', age=25, job='dev')
# Prints {'name': 'Bob', 'age': 25, 'job': 'dev'}
```

Return Value from Function

- To return a value from a function, simply use a return statement.
- Once a return statement is executed, nothing else in the function body is executed.

```
# Return sum of two values
def sum(a, b):
    return a + b

x = sum(3, 4)
print(x)
# Prints 7
```

- Python has the ability to return multiple values, something missing from many other languages.
- You can do this by separating return values with a comma

```
# Return addition and subtraction in a tuple
def func(a, b):
    return a+b, a-b

result = func(3, 2)

print(result)
# Prints (5, 1)
```

The Docstring

- You can attach documentation to a function definition by including a string literal just after the function header.
- Docstrings are usually triple quoted to allow for multi-line descriptions.

```
def hello():
    """This function prints
       message on the screen"""
    print('Hello, World!')
```

```
# Print docstring in rich format
help(hello)

# Help on function hello in module __main__:
# hello()
#     This function prints
#     message on the screen
```

```
# Print docstring in a raw format
print(hello.__doc__)

# Prints This function prints message on the screen
```

Function Operations

- Nested Functions

```
def outer(a, b):
    def inner(c, d):
        return c + d
    return inner(a, b)

result = outer(2, 4)

print(result)
# Prints 6
```

- Recursion

```
def countdown(num):
    if num <= 0:
        print('Stop')
    else:
        print(num)
        countdown(num-1)

countdown(5)
# Prints 5
# Prints 4
# Prints 3
# Prints 2
# Prints 1
# Prints Stop
```

- Assigning function to variable

```
def hello():
    print('Hello, World!')

hi = hello
hi()
# Prints Hello, World!
```

Variable Scopes

- Not all variables are accessible from all parts of our program.
- The part of the program where the variable is accessible is called its “scope” and is determined by where the variable is declared.
- Python has three different variable scopes:
  - Local scope
  - Global scope
  - Enclosing scope

Local Scope

```
def myfunc():
    x = 42      # local scope x
    print(x)

myfunc()        # prints 42
```

```
def myfunc():
    x = 42      # local scope x

myfunc()
print(x)        # Triggers NameError: x does not exist
```

Global Scope

```
x = 42          # global scope x

def myfunc():
    print(x)    # x is 42 inside def

myfunc()
print(x)        # x is 42 outside def
```

```
x = 42          # global scope x
def myfunc():
    x = 0
    print(x)    # local x is 0

myfunc()
print(x)        # global x is still 42
```

```
verbose = True

def op1():
    if verbose:
        print('Running operation 1')
```

```
x = 42          # global scope x
def myfunc():
    global x    # declare x global
    x = 0
    print(x)    # global x is now 0

myfunc()
print(x)        # x is 0
```

Enclosing Scope

```
# enclosing function
def f1():
    x = 42
    # nested function
    def f2():
        x = 0
        print(x)    # x is 0
    f2()
    print(x)        # x is still 42

f1()
```

```
# enclosing function
def f1():
    x = 42
    # nested function
    def f2():
        nonlocal x
        x = 0
        print(x)    # x is now 0
    f2()
    print(x)        # x remains 0

f1()
```

Generator Functions

- Generators are iterable functions.
- Generate values once at a time from a given sequence, instead of giving entire sequence.
- yield keyword is used to return values from generator functions.
- We need to call next() function in order to get values one by one from generator functions.
- After returning all values one by one next() function raises StopIteration error, indicating all values are taken out.
- Example:

```
def gnrtrFunc():
  for i in range(1,11):
    yield i**2

values=gnrtrFunc()
print(type(values))
print(values)

print(next(values))

for i in values:
  print(i)
```

```
<class 'generator'>
<generator object gnrtrFunc at 0x000001A93D9FD468>
1
4
9
16
25
36
49
64
81
100
>>>
```

Lambda Expression

- A lambda is simply a way to define a function in Python. They are sometimes known as lambda expressions or lambda operators.
- The lambda functions are anonymous. Meaning, they are functions that do not need to be named. They are used to create small one-line functions in cases where a normal function would be an overkill
- Have any number of arguments but have only one expression.
- Always return an expression

lambda parameters: expression

Lambda Functions contd.

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

print(doubler(2))
# Prints 4

print(doubler(5))
# Prints 10
```

```
doubler = lambda x: x*2

print(doubler(2))
# Prints 4

print(doubler(5))
# Prints 10
```

```
evenOdd = (lambda x:
           'odd' if x%2 else 'even')

print(evenOdd(2))
# Prints even

print(evenOdd(3))
# Prints odd
```

```
# A lambda function that adds three values
add = lambda x, y, z: x+y+z
print(add(2, 5, 10))
# Prints 17
```

Lambda with map, filter, reduce, sorted

```
# Double each item of the list
L = [1, 2, 3, 4, 5, 6]
doubler = map(lambda x: x*2, L)
print(list(doubler))
# Prints [2, 4, 6, 8, 10, 12]
```

```
from functools import reduce

L = [10, 20, 30, 40]
result = reduce(lambda a, b: a + b, L)
print(result)
# Prints 100
```

```
# Filter the values above 18
age = [5, 11, 16, 19, 24, 42]
adults = filter(lambda x: x > 18, age)
print(list(adults))
# Prints [19, 24, 42]
```

```
# Sort the list of taples by the age of students
L = [('Sam', 35),
    ('Max', 25),
    ('Bob', 30)]
x = sorted(L, key=lambda student: student[1])
print(x)
# Prints [('Max', 25), ('Bob', 30), ('Sam', 35)]
```

Decorator Functions

- Sometimes you want to modify an existing function without changing its source code. A common example is adding extra processing (e.g. logging, timing, etc.) to the function.
- That's where decorators come in.
- A decorator is a function that accepts a function as input and returns a new function as output, allowing you to extend the behavior of the function without explicitly modifying it.
- In Python, a function can be:
  - Assigned to a variable
  - Passed as argument
  - Defined inside another function
  - Returned from a function

Function Decorators

- Syntax

```
@funcDcrtr
def dcrtrFunc():
        print(“func”)
```

- Is equivalent to

```
def dcrtrFunc():
    print(“func”)
dcrtrFunc = funcDcrtr(dcrtrFunc)
```

```
def decorate_it(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper

def hello():
    print("Hello world")

hello = decorate_it(hello)

hello()
# Prints Before function call
# Prints Hello world
# Prints After function call
```

Modules & Packages

Python Functions

Creating Python Module

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

```
>>> import fibo
```

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Python Module Import

- Module is a collection of functions, classes and variables packaged in a file.
- Enables code reusability.
- Can be brought in using import, from, as keywords.
- **Syntax1**

```
import <ModuleName>
#this will import the source file ModuleName in another file.
```

- **Syntax2**

```
from <ModuleName> import name1[,name2[,nameN]]
#this will import specific attributes from a ModuleName into current file.
```

- **Syntax3**

```
from <ModuleName> import *
#this will import all attributes from a ModuleName into current file.
```

- **Syntax4**

```
import <ModuleName> as <NewName>
#this will import ModuleName renamed as NewName
```

Setting Module Search Path

```
import sys
sys.path.append('/home/test/')

import calculation
print(calculation.add(1,2))
```

Installing Python Modules

- `python –m pip install <module>`
- Or
- `pip install <module>`

- For multiple python versions, specify python version
- To install PIP (Python Package Installer)
  - Download `get-pip.py`
  - Run
    - `python get-pip.py`

Common Modules

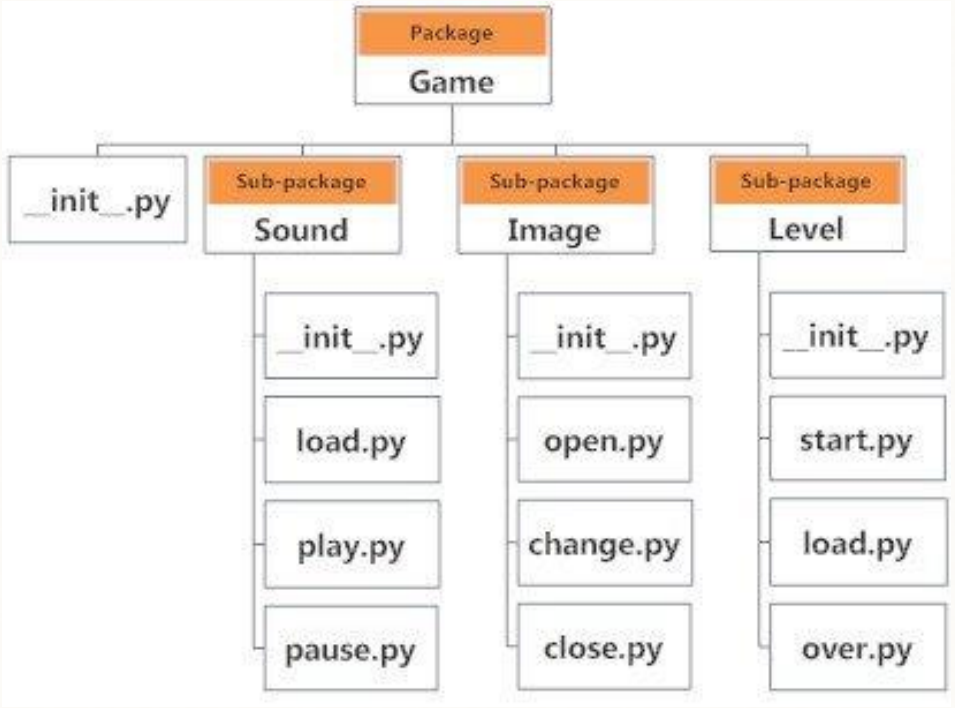
- os
- sys
- math
- json
- csv
- numpy
- pandas
- sklearn
- tkinter

- Using dir() function

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
 '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
 '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework',
 '_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
 'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
 'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing',
 'callstats', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
 'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
 'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',
 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'pycache_prefix',
 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',
 'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', 'version_info',
 'warnoptions']
```

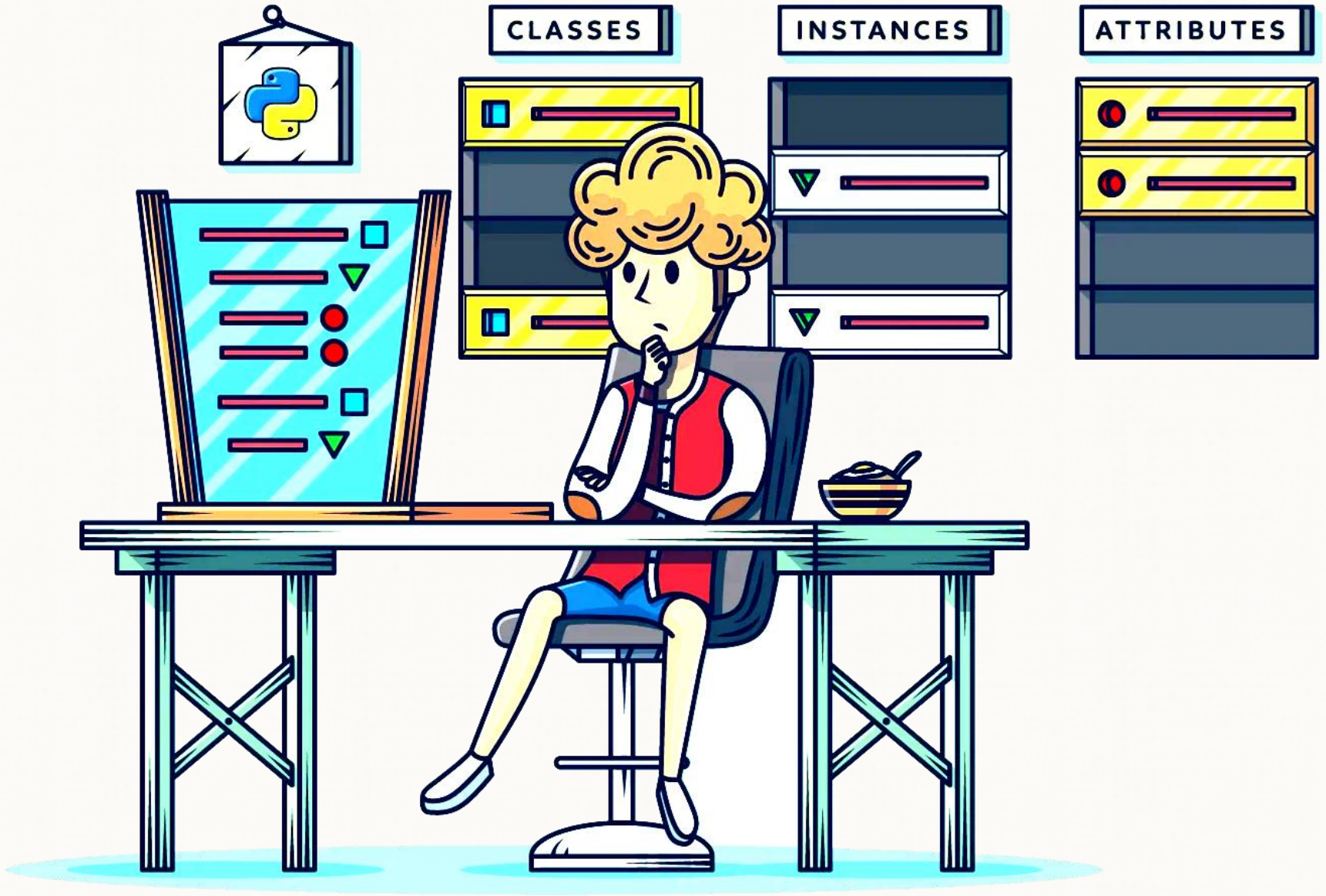
Python Packages

- Packages are a way of structuring Python's module namespace by using "dotted module names".
- Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory. Analogous to this, Python has packages for directories and modules for files.
- As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear.
- Similarly, as a directory can contain subdirectories and files, a Python package can have sub-packages and modules.
- A directory must contain a file named __init__.py in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.



Object Oriented Programming

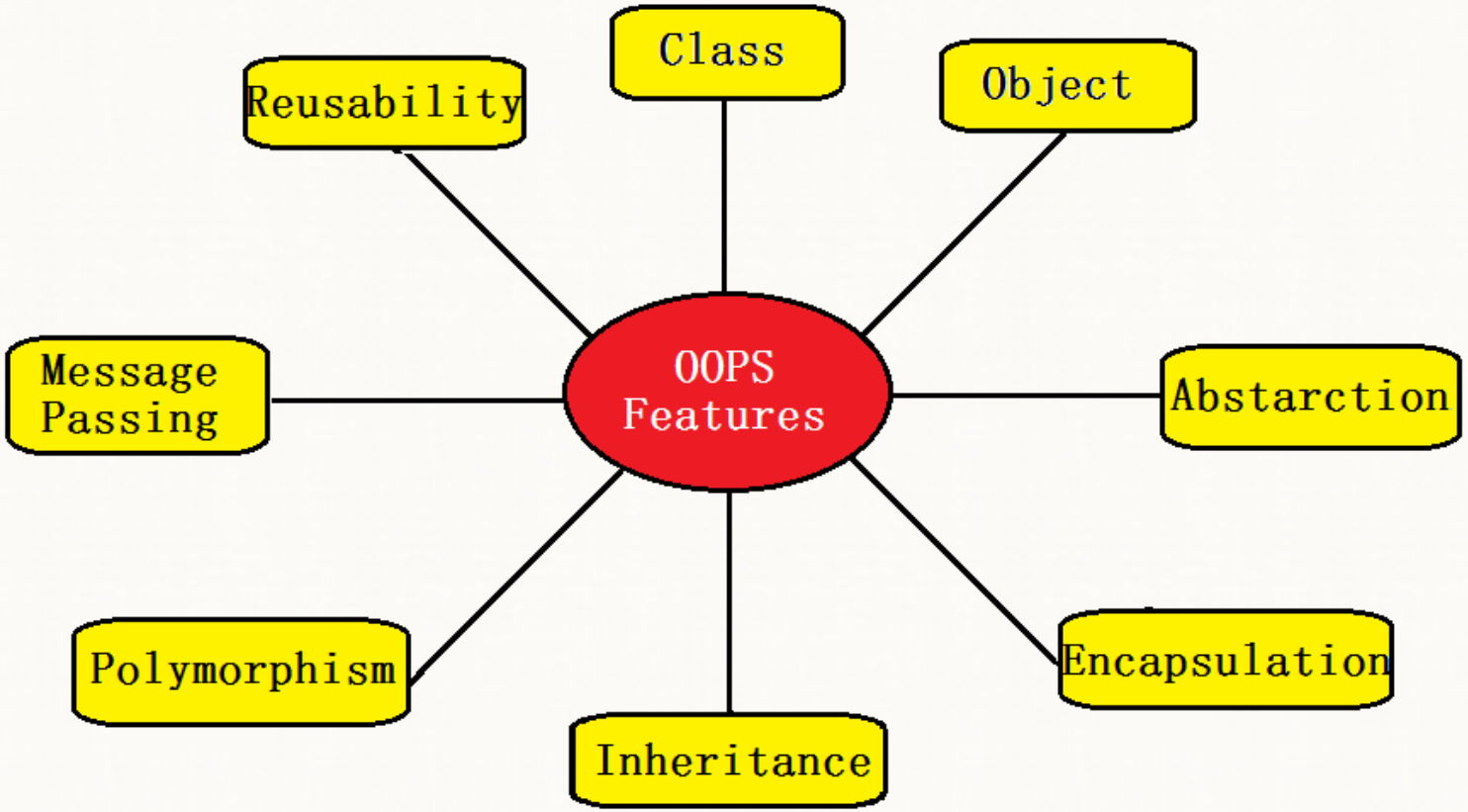
Python Functions


CLASSES
INSTANCES
ATTRIBUTES

OOP

- **Object-oriented Programming**, or OOP for short, is a programming paradigm which provides a means of structuring programs so that properties and behaviors are bundled into individual objects.
- Object-oriented programming is an approach for modeling concrete, real-world things like cars as well as relations between things like companies and employees, students and teachers, etc.
- OOP models real-world entities as software objects, which have some data associated with them and can perform certain functions.
- Another common programming paradigm is ***procedural programming*** which structures a program like a recipe in that it provides a set of steps, in the form of functions and code blocks, which flow sequentially in order to complete a task

OOP Features



OOP – Terminologies

- **Class:** It is a kind of template, which comprises of multiple attributes and methods (called as its members also)

```
class Parrot:
        pass
```

- **Object:** Its an instance (handle) of a class. With this, we can access all the features of any class.

```
obj = Parrot()
```

- **Methods:** Methods are functions defined inside the body of a class. They are used to define the behaviors of an object

```
def function(self):
        pass
```

- **Inheritance:** Inheritance is a way of creating new class for using details of existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class)

OOP – Terminologies contd.

- Encapsulation: Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix i.e single “ _ “ or double “ __“.
- **Polymorphism:** Polymorphism is an ability (in OOP) to use common interface for multiple form (data types).Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However we could use same method to color any shape. This concept is called Polymorphism.

Defining a Class in Python

```
class Parrot():
        print('Class created')
        a = 20
        b = 30

obj = Parrot()
print(obj)
print(obj.a)
print(obj.b)
```

Defining a Method in Python

```
class Parrot():
        print('Class created')
        a = 20
        b = 30
        def function(self):
                print("Great")

obj = Parrot()
obj.function()
```

Attributes in Classes

- Instance Attributes
  - These are the ones which have separate values for separate objects. (Initialized with default values in all, if given)
  - self.name
- Class Attributes
  - These are common for all class members, objects. If one tries to change them, they get changed for all.
  - ClassName.name

Methods in Classes

- User Defined Methods
  - The methods which we define by our own for custom purposes
  - E.g.

```
def function (self):
    pass
```

- Dunder Functions (Magic Functions)
  - The pre-built name of methods, which actually covered under double underscores as leading and trailing characters

```
def __init__(self):
    print('Hi')
```

```
class Employee: #class created
  id = 0    #property (variable)
  name = "" #property (variable)
  def createEmp(self,id,name): #behavior (method)
    self.id = id
    self.name = name

  def showInfo(self):  #behavior (method)
    print("Id:",self.id)
    print("Name:",self.name)


emp1 = Employee() #object1
emp2 = Employee() #object2

emp1.createEmp(1,"ABC")  #access behavior
emp2.createEmp(2,"XYZ")  #access behavior

emp1.showInfo()  #access behavior
emp2.showInfo()  #access behavior

emp1.name = "PQR"  #access property

emp1.showInfo()  #access behavior
emp2.showInfo()  #access behavior
```

```
Python 3.6.5 (v3.6.5
4)] on win32
Type "copyright", "
>>>
================== R
Id: 1
Name: ABC
Id: 2
Name: XYZ
Id: 1
Name: PQR
Id: 2
Name: XYZ
>>>
```

Class Attributes

File Edit Format Run Options Window Help

```
class Student:
  count = 0  #static variable
  def __init__(self):
    Student.count += 1  #must be accessed using class name

s1 = Student()
s2 = Student()
s3 = Student()

print("Number of students:",Student.count)
```

File Edit Shell Debug Options Win

```
Python 3.6.5 (v3.6.5:f5
4)] on win32
Type "copyright", "cred
>>>
================== REST
Number of students: 3
>>>
```

Python Dunder Functions

- Constructors & Destructors: __init__, __del__
- Iteration & Length: __getitem__, __len__
- Method Invocation: __call__
- Strings Representation: __str__, __repr__
- Operator Overloading: __eq__, __lt__, __add__, __sub__, __mul__, …

Constructors

- It is a special type of method which is used to initialize the instances of the class.
- Constructor is executed with creation of the objects.
- It also verify that there are enough resources for the object to perform any start-up task.
- To create constructor __init__ method is used. We can pass any number of arguments to create and initialize objects to constructor.

```
File  Edit  Format  Run  Options  Window  Help
class Employee: #class created
  id = 0     #property (variable)
  name = "" #property (variable)

  def __init__(self,id,name):  #constructor
    self.id = id
    self.name = name

  def showInfo(self):  #behavior (method)
    print("Id:",self.id)
    print("Name:",self.name)


emp1 = Employee(1,"ABC") #object1
emp2 = Employee(2,"XYZ") #object2

emp1.showInfo()  #access behavior
emp2.showInfo()  #access behavior

emp1.name = "PQR"  #access property

emp1.showInfo()  #access behavior
emp2.showInfo()  #access behavior
```

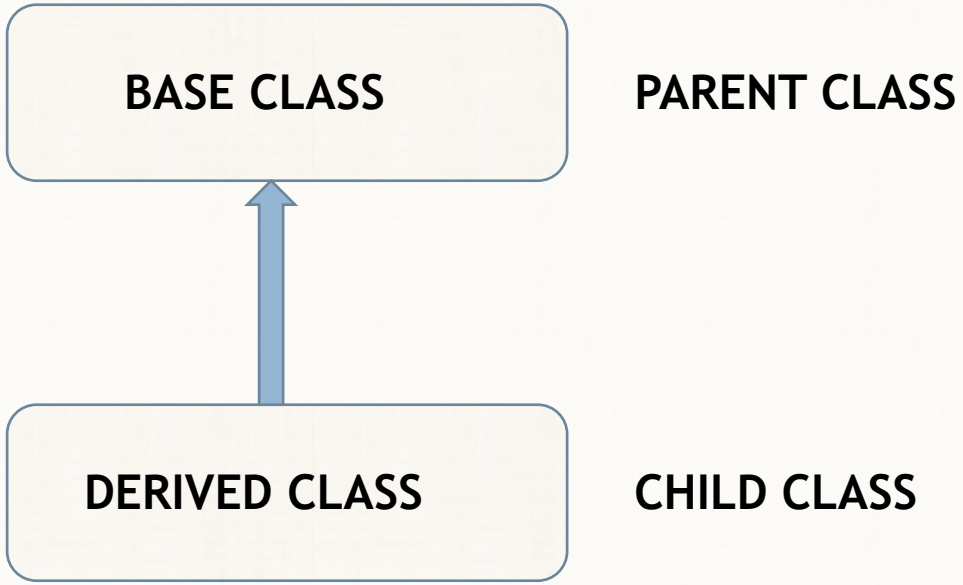
```
File  Edit  Shell  Debug  Optio
Python 3.6.5 (v3.6.
4)] on win32
Type "copyright", "
>>>
==================
Id: 1
Name: ABC
Id: 2
Name: XYZ
Id: 1
Name: PQR
Id: 2
Name: XYZ
>>>
```

Inheritance

- Provide code reusability because we can use existing class to create a new class.
- The child class acquires properties and can access all data members and functions in the parent
- Syntax

```
class derived-class(base-class):
    #block of class
```

- Types:
  - Single Level
  - Multilevel
  - Multiple
  - Hybrid



File Edit Format Run Options Window

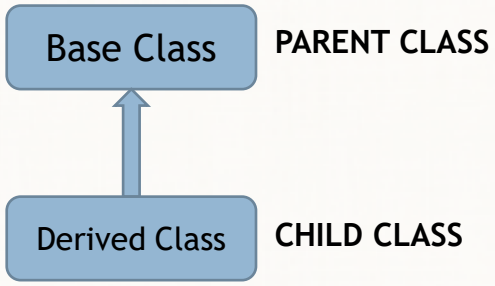
```
class Animal:
  no_of_legs = 4
  def speak(self):
    print("Animal class")

class Dog(Animal):
  def bark(self):
    print("Dog Barking")

d = Dog()
print(d.no_of_legs)
d.speak()
d.bark()

a= Animal()
print(a.no_of_legs)
a.speak()
a.bark()
```

File Edit Shell Debug Options Window Help

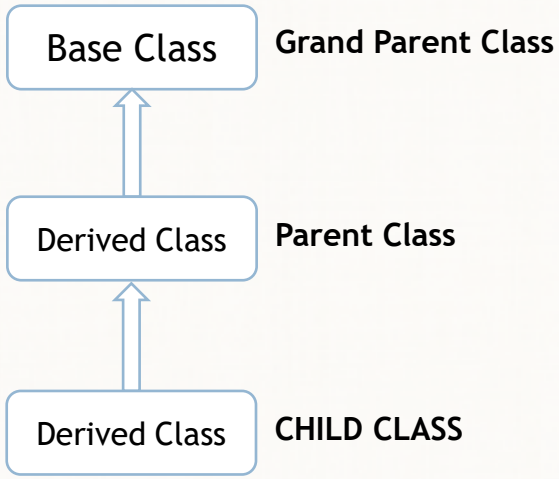
```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.
4)] on win32
Type "copyright", "credits" or "license()" for more information
>>>
================== RESTART:                                    ===
4
Animal class
Dog Barking
4
Animal class
Traceback (most recent call last):
  File "C:\Users\Himanshu\Desktop\vv.py", line 18, in <module>
    a.bark()
AttributeError: 'Animal' object has no attribute 'bark'
>>>
```

Types of Inheritance

- Single Level

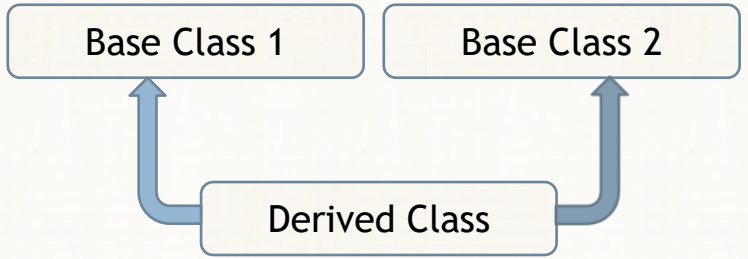


- Multilevel

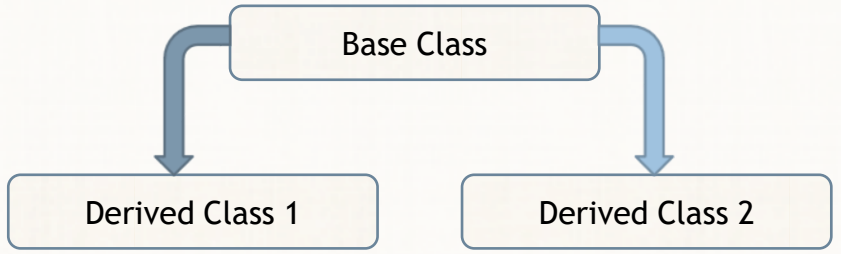


Types of Inheritance

- Multiple



- Hierarchical



Polymorphism

- Polymorphism can be achieved using method overriding.
- Parent class method is defined in the child class with specific implementation.

File Edit Format Run Options Window Help

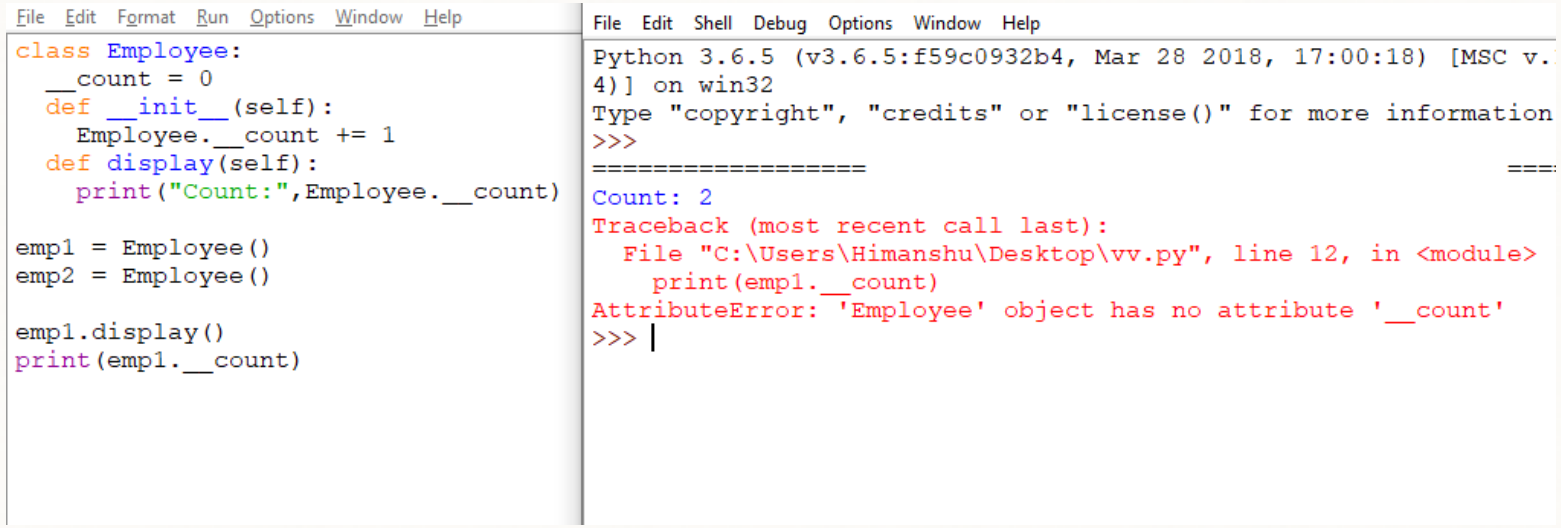
```
class Bank:
  def getroi(self):
    return 4.5

class Kotak(Bank):
  def getroi(self):
    return 6.5

class ICICI(Bank):
  def getroi(self):
    return 6.0

b1 = Bank()
b2 = Kotak()
b3 = ICICI()

print("ROI for Bank:",b1.getroi())
print("ROI for Kotak:",b2.getroi())
print("ROI for ICICI:",b3.getroi())
```

File Edit Shell Debug Options Win

```
Python 3.6.5 (v3.6.5:f5
4)] on win32
Type "copyright", "cred
>>>
================== REST
ROI for Bank: 4.5
ROI for Kotak: 6.5
ROI for ICICI: 6.0
>>>
```

Abstraction (Data Hiding)

- This can be done by '__' double under scores
- Although we can access our elements by object, obj._ClassName__data



Errors & Exception Handling

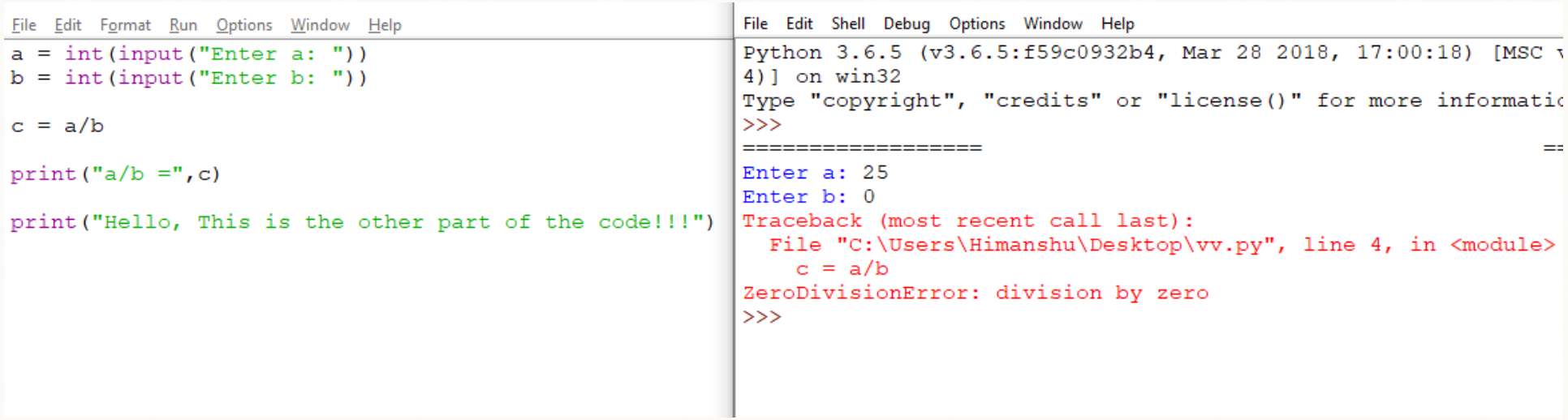
Python Functions

Python Errors & Exceptions

- An exception can be defined as an abnormal condition in a program resulting in the flow of the program.
- It causes the program to halt the execution.
- So exception handling is a way to deal with this problem. So that other part of the code can be executed without any disruption.
- Common Exceptions
  - ZeroDivisionError – Occurs when a number is divided by 0.
  - NameError – Occurs when a name is not found (local or global).
  - IndentationError – Occurs when incorrect indentation is given.
  - IOError – Occurs when Input Output operation fails.
  - EOFError – Occurs when end of file is reached, and yet operations are being performed.

Python Exception Handling contd.

- Problem without handling exception
- Here, we have entered b=0.
- It raises ZeroDivisionError at c = a/b.
- It causes program to entered in halt condition.
- So in order to handle this situation, We have exception handling



Python Exception Handling contd.

- We have following keywords to handle exceptions, so that we can deal with exceptions in several ways
  - try
  - except
  - else
  - finally
  - raise
- The try and except keywords

```
try:
      Test This Code For An Exception
except:
      Run This Code If An Exception Occurs
```

- We can have multiple except blocks associated with one try block.

File Edit Format Run Options Window Help

```
a = int(input("Enter a: "))
b = int(input("Enter b: "))

try:
  c = a/b
except ZeroDivisionError:
  print("Denominator can't be zero, Enter again!!!")
  b = int(input("Enter b: "))
  c = a/b

print("a/b =",c)

print("Hello, This is the other part of the code!!!")
```

File Edit Shell Debug Options Window Help

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018,
4)] on win32
Type "copyright", "credits" or "license()" fo
>>>
==================
Enter a: 25
Enter b: 0
Denominator can't be zero, Enter again!!!
Enter b: 12
a/b = 2.0833333333333335
Hello, This is the other part of the code!!!
>>>
```

Python Exception Handling contd.

- The else keyword

```
try:
        #test code
except Exception:
        #handle exception, if any
else:
        #do this if no exception raised
```

```
a = int(input("Enter a: "))
b = int(input("Enter b: "))

try:
  c = a/b
except ZeroDivisionError:
  print("Denominator can't be zero, Enter again!!!")
  b = int(input("Enter b: "))
  c = a/b
else:
  print("Well Done, No exception!!!")

print("a/b =",c)

print("Hello, This is the other part of the code!!!")
```

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018,
4)] on win32
Type "copyright", "credits" or "license()" for
>>>
==================
Enter a: 25
Enter b: 12
Well Done, No exception!!!
a/b = 2.0833333333333335
Hello, This is the other part of the code!!!
>>>
```

Python Exception Handling contd.

- The **finally** keyword

```
try:
        #test code
except Exception:
        #handle exception, if any
finally:
        #always run this code
```

```
File  Edit  Format  Run  Options  Window  Help
a = int(input("Enter a: "))
b = int(input("Enter b: "))

try:
  c = a/b
except ZeroDivisionError:
  print("Denominator can't be zero, Enter again!!!")
  b = int(input("Enter b: "))
  c = a/b
finally:
  print("I will always run!!!")

print("a/b =",c)

print("Hello, This is the other part of the code!!!")
```

```
File  Edit  Shell  Debug  Options  Window  Help
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 1
4)] on win32
Type "copyright", "credits" or "license()" for
>>>
==================
Enter a: 25
Enter b: 0
Denominator can't be zero, Enter again!!!
Enter b: 8
I will always run!!!
a/b = 3.125
Hello, This is the other part of the code!!!
>>>
================== RESTART: C:\Users\Himanshu\I
Enter a: 25
Enter b: 8
I will always run!!!
a/b = 3.125
Hello, This is the other part of the code!!!
>>>
```

Python Exception Handling contd.

- The raise keyword :- Exception can be raised using following syntax-
- `raise Exception_class, <value>`

File Edit Format Run Options Window Help

```
try:
  age = int(input("Enter the age: "))
  if age<18:
    raise ValueError
  else:
    print("The age is valid!!!")
except ValueError:
  print("The age is not valid!!!")
```

File Edit Shell Debug Options Window

```
Python 3.6.5 (v3.6.5:f59c09
4)] on win32
Type "copyright", "credits"
>>>
================== RESTART:
Enter the age: 15
The age is not valid!!!
>>>
================== RESTART:
Enter the age: 18
The age is valid!!!
>>>
```

File Handling

Python Functions

Simple File Handling

- Python can handle files with following commands

```
with open('myfile', 'r') as f:
     r = f.read()
     f.close()
with open('myfile', 'w+') as f:
     f.write('Mydata String')
     f.close()
with open('myfile', 'a+') as f:
     f.write('More data strings')
     f.close()
```

File Modes

MODE	DESCRIPTION
r	Read only mode, open text file, pointer location beginning, file must exist.
rb	Read only mode, open binary file, pointer location beginning, file must exist.
r+	Read & Write mode, open text file, pointer location beginning, file must exist.
rb+	Read & Write mode, open binary file, pointer location beginning, file must exist.
w	Write only mode, open text file, pointer location beginning, file created if not exist.
wb	Write only mode, open binary file, pointer location beginning, file created if not exist.
w+	Write and read mode, open text file, pointer location beginning, file created if not exist.
wb+	Write and read mode, open binary file, pointer location beginning, file created if not exist.
a	Append only mode, open text file, pointer location end of file, file created if not exist.
ab	Append only mode, open binary file, pointer location end of file, file created if not exist.
a+	Append and read mode, open text file, pointer location end of file, file created if not exist.
ab+	Append and read mode, open binary file, pointer location end of file, file created if not exist.
x	Create a new file.

THANK YOU