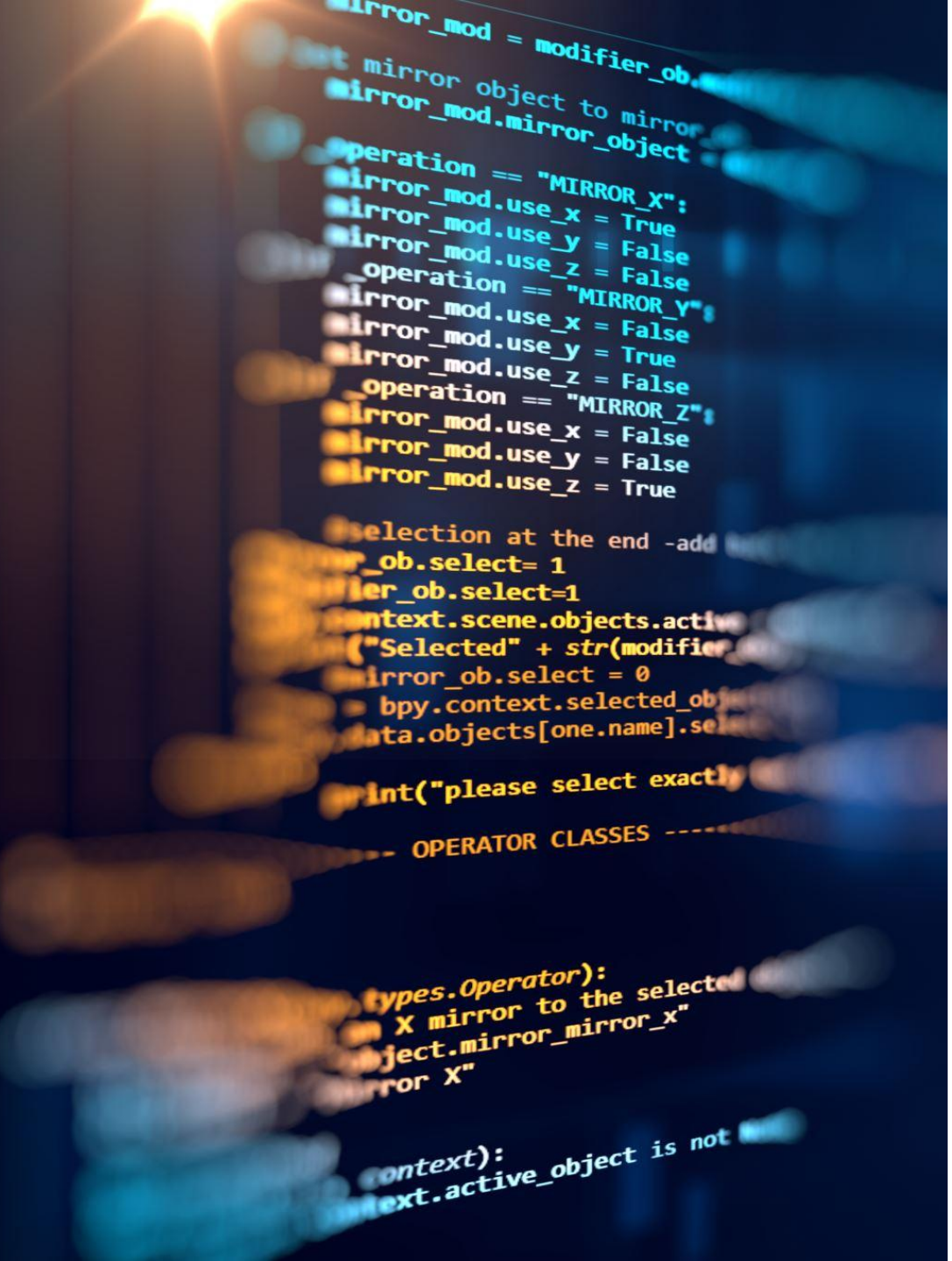
PYTHON BASIC

BY AMAR PANCHAL



Career Credentials

PYTHON IS

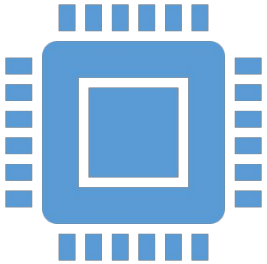
- Python is a high-level, interpreted, interactive and object-oriented scripting language. Python was designed to be highly readable which uses English keywords frequently whereas other languages use punctuation and it has fewer syntactical constructions than other languages. Created by Guido van Rossum and first released in 1991

Career Credentials

History of Python

- December 1989 I Implementation by Guido van Rossum as successor of ABC to provide exception handling
- February 1991 I Python 0.9.0 had classes with inheritance, exception handling, functions, and the core datatypes
- January 1994 I Version 1.0 has functional programming features
- October 2000 I Python 2.0 brings garbage collections
- Python 2.2 improves Python's types to be purely object oriented
- December 2008 I Python 3 I Reduce feature duplication by removing old ways of doing things.
- **Not backwards compatible with Python 2.x**

History of Python



Modules

Reusable pieces of software

Can be written by any Python developer

Extend Python's capabilities

Python Web site at www.python.org

Primary distribution center for Python source code, modules and documentation

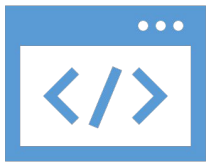
VERSION PROBLEM

PYTHON 2.X	PYTHON 3.X
started in 2000 and slated to lose all support in 2020	Started in 2008 and seen as future of Python
Absence of new modules	Newer modules like __future__ and asy nio
print keyword	print() is a function
ASCII and UNICODE support Use u for Unicode u 'Hello‘ else byte sequence	UNICODE default All strings are now Unicode b used for byte sequence
Division With Integers 5 / 2=2	Division With Integers 5 / 2=2.5
raw_input() function and an input()	Only input()

PYTHON IS GREAT BECAUSE...

Portable

Your program will run if the interpreter works and the modules are installed



Efficient

Rich language features built in

Simple syntax for ease of reading

Focus shifts to “algorithm”

Flexible

Imperative

Object-oriented

Functional

PYTHON IS GREAT BECAUSE...

Extendible

Plenty of easy-to-use modules

If you can imagine it, then someone has probably developed a module for it

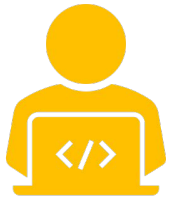
Your program can adjust the way the interpreter works

Python syntax

Extremely simplified syntax

Nearly devoid of special characters

Intended to be nearly English

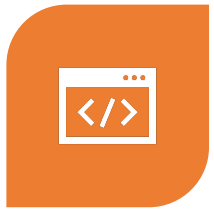


Dynamic types

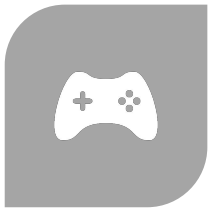
It is the responsibility of the programmer

Still “strongly typed

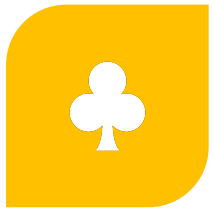
INDUSTRIAL USE OF PYTHON



WEB DEVELOPMENT



GAMES

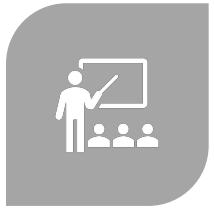


GRAPHICS

FINANCIAL

SCIENCE

ELECTRONIC DESIGN AUTOMATION

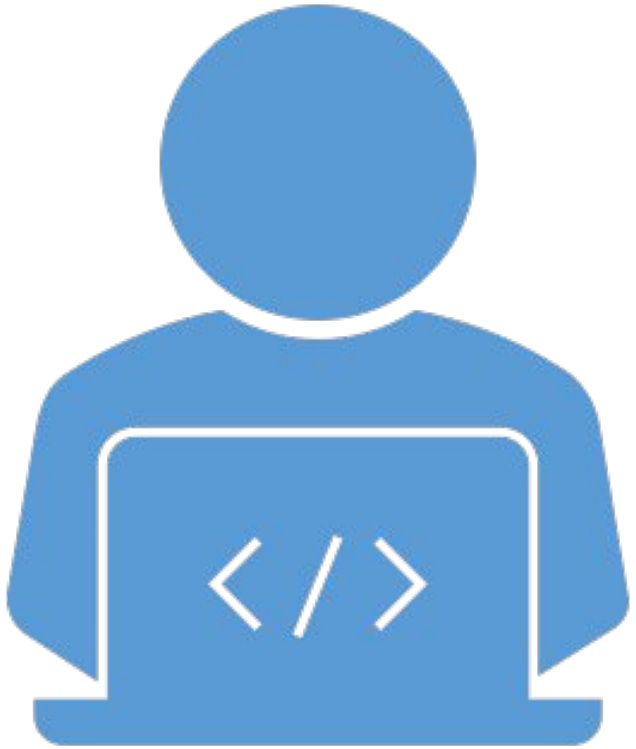


EDUCATION

BUSINESS SOFTWARE

INTERACTIVE MODE

Running



- **We assume that you have the Python interpreter set in PATH variable. Now, try to run this program as follows-**
- **On Linux: $ python test.py**

This produces the following result-

- **Hello, Python!**
- **On Windows**
- **C:\Python37>Python test.py**

This produces the following result-

- **Hello, Python!**

Python Identifiers

An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language

Naming conventions for python identifiers

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strong private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language defined special name.

Key Words

and	exec	Not
as	finally	or
assert	for	pass
break	from	print
class	global	raise
continue	if	return
def	import	try
del	in	while
elif	is	with
else	lambda	yield
except		

Lines and Indentation

- **Python does not use braces({}) to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by l**
- **if True:**
- **print ("True")**
- **else:**
- **print ("False") ine indentation, which is rigidly enforced.**

Multi-Line Statements

- Statements in Python typically end with a new line. Python, however, allows the use of the line continuation character (\) to denote that the line should continue.

```
total = item_one + \
        item_two + \
        item_three
```

The statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example-

```
days = ['Monday', 'Tuesday', 'Wednesday',
            'Thursday', 'Friday‘]
```

Quotation in Python

- Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

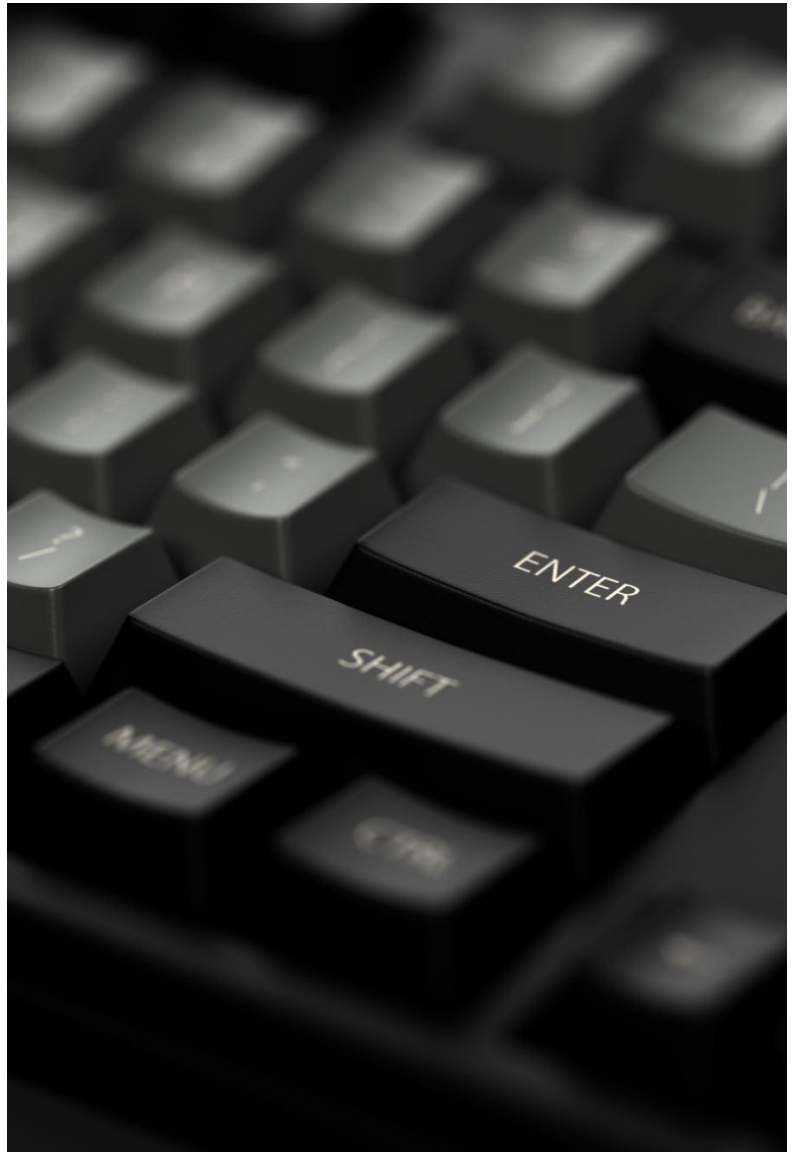
Comments in Python

- A hash sign (#) that is not inside a string literal is the beginning of a comment. All characters after the #, up to the end of the physical line, are part of the comment and the Python interpreter ignores them.

- # First comment
- print ("Hello, Python!") # second comment

Waiting for the User

- input("\Press the enter key to exit.")



Multiple Statements on a Single Line

- The semicolon (;) allows multiple statements on a single line given that no statement starts a new code block.
- Example

Suites

- Groups of individual statements, which make a single code block are called suites in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite
- Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite.

```
if expression :
   suite
elif expression :
   suite
else :
   suite
```

Your First Python Program

- At the prompt (>>>) type:

```
print "Game Over"
```

  - Press [Enter]

- Very straightforward
  - You could have guessed what this does without knowing Python!

Career Credentials

PYTHON-VARIABLES

BY AMAR PANCHAL

Assigning Values to Variables

- Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.
- Python allows you to assign a single value to several variables simultaneously.
- EXAMPLE
- Special case also
  - a, b, c = 24, 36, “AMAR”

Standard Data Types

- Python has five standard data types-
  - Numbers
  - Boolean
  - String
  - List
  - Tuple
  - Dictionary
  - Set

Numbers

- int (signed integers)
- float (floating point real values)
- complex (complex numbers)

Lists

- A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One of the differences between them is that all the items belonging to a list can be of different data type.
- The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.

```
list = [ ‘amp', 4624 , 3.14, ‘python', 0.07 ]

tinylist = [420, ‘abcd’]

print (list)

print (list[0])

print (list[1:3])

print (list[2:])

print (tinylist * 2)

print (list + tinylist)
```

LIST

Function	Description
all()	Return True if all elements of the list are true (or if the list is empty).
any()	Return True if any element of the list is true. If the list is empty, return False.
enumerate()	Return an enumerate object. It contains the index and value of all the items of list as a tuple.
len()	Return the length (the number of items) in the list.
list()	Convert an iterable (tuple, string, set, dictionary) to a list.
max()	Return the largest item in the list.
min()	Return the smallest item in the list
sorted()	Return a new sorted list (does not sort the list itself).
sum()	Return the sum of all elements in the list.

- fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']

```
>>> fruits.count('apple')
>>> fruits.count('tangerine')
>>> fruits.index('banana')
>>> fruits.index('banana', 4)
>>> fruits.reverse()
>>> fruits
>>> fruits.append('grape')
>>> fruits
```

Tuples

- A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parenthesis.
- The main difference between lists and tuples are – Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists.

```
tuples = ( ‘amp', 4624 , 3.14, ‘python', 0.07 )
tinylist = (420, ‘abcd’)
print (list)
print (list[0])
print (list[1:3])
print (list[2:])
print (tinylist * 2)
print (list + tinylist)
```

del

The del statement

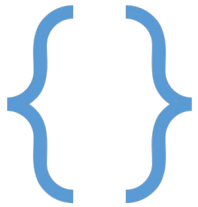
CODE:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a

>>> del a[2:4]
>>> a

>>> del a[:]
>>> a

>>> del a
>>>print(a)
```

Dictionary

- Python's dictionaries are kind of hash-table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.
- Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).
- dict = {}
- dict['one'] = "This is one"
- dict[2] = "This is two"
- tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
- print (dict['one'])
- print (dict[2])
- print (tinydict)
- print (tinydict.keys())
- print (tinydict.values())

Method	Description
clear()	Remove all items form the dictionary.
copy()	Return a shallow copy of the dictionary.
fromkeys(seq[, v])	Return a new dictionary with keys from seq and value equal to v(defaults to None).
get(key[,d])	Return the value of key. If key doesnot exit, return d (defaults to None).
items()	Return a new view of the dictionary's items (key, value).
keys()	Return a new view of the dictionary's keys.
pop(key[,d])	Remove the item with key and return its value or d if key is not found. If d is not provided and key is not found, raises KeyError.
popitem()	Remove and return an arbitary item (key, value). Raises KeyError if the dictionary is empty.
setdefault(key[,d])	If key is in the dictionary, return its value. If not, insert key with a value of d and return d (defaults to None).
update([other])	Update the dictionary with the key/value pairs from other, overwriting existing keys.
values()	Return a new view of the dictionary's values



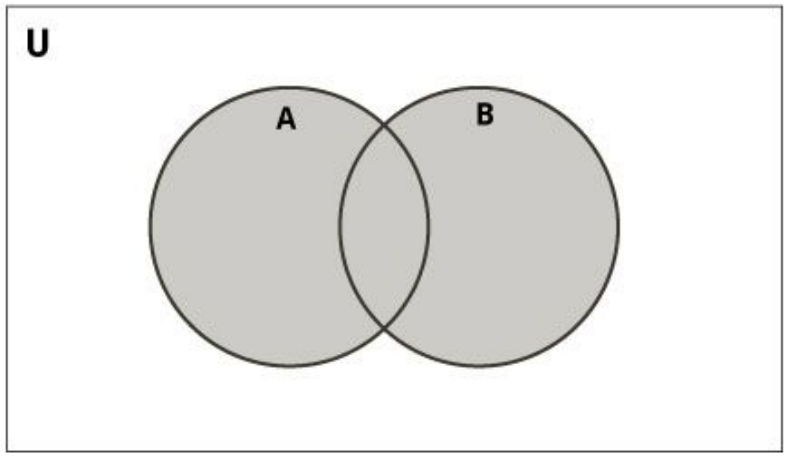
Set

- A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.

Python Set Methods

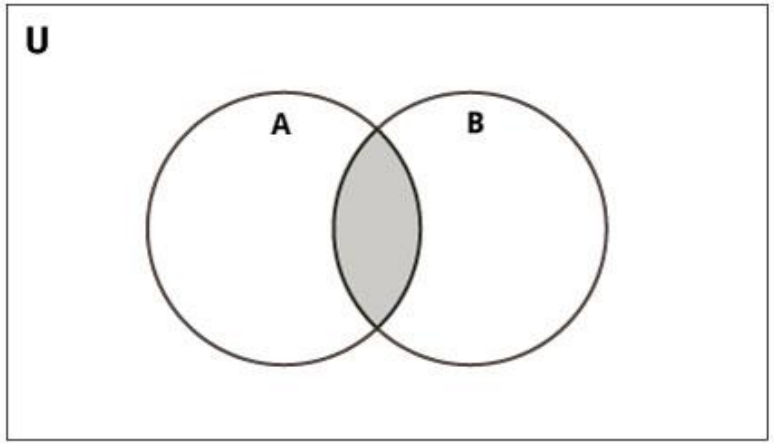
Method	Description
add()	Add an element to a set
clear()	Remove all elements form a set
copy()	Return a shallow copy of a set
difference()	Return the difference of two or more sets as a new set
difference_update()	Remove all elements of another set from this set
discard()	Remove an element from set if it is a member. (Do nothing if the element is not in set)
intersection()	Return the intersection of two sets as a new set
intersection_update()	Update the set with the intersection of itself and another
isdisjoint()	Return True if two sets have a null intersection
issubset()	Return True if another set contains this set
issuperset()	Return True if this set contains another set
pop()	Remove and return an arbitary set element. Raise KeyErrorif the set is empty
remove()	Remove an element from a set. If the element is not a member, raise a KeyError
symmetric_difference()	Return the symmetric difference of two sets as a new set
symmetric_difference_update()	Update a set with the symmetric difference of itself and another
union()	Return the union of sets in a new set
update()	Update a set with the union of itself and others

Set Union



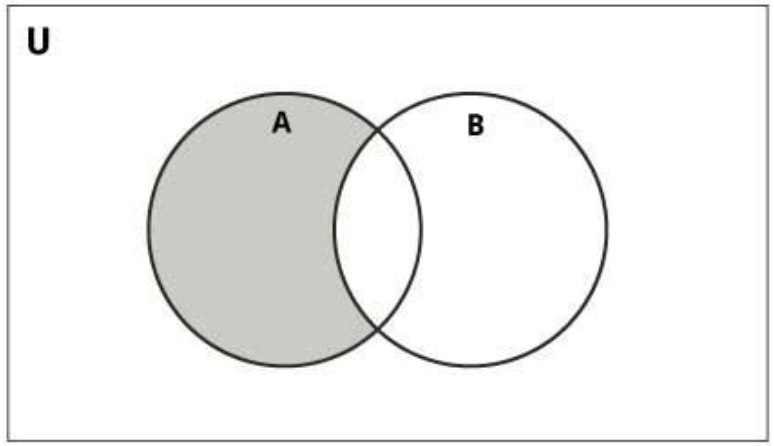

```
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
# use | operator
print(A | B)
```

Set Intersection



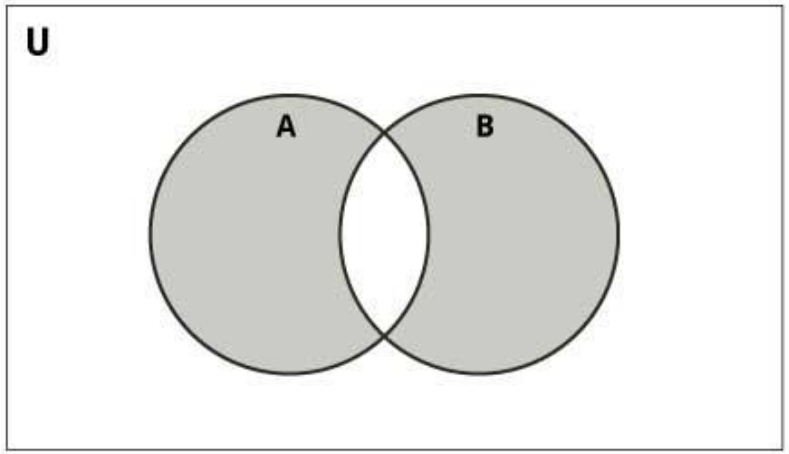
- # initialize A and B
- A = {1, 2, 3, 4, 5}
- B = {4, 5, 6, 7, 8}
- # use & operator#
- print(A & B)

Set Difference



- # initialize A and B
- A = {1, 2, 3, 4, 5}
- B = {4, 5, 6, 7, 8}
- print(A - B)

Set Symmetric Difference



- # initialize A and B
- A = {1, 2, 3, 4, 5}
- B = {4, 5, 6, 7, 8}
- print(A ^ B)

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

Data Type Conversion

Function	Description
int(x [,base])	Converts x to an integer. The base specifies the base if x is as string.
float(x)	Converts x to a floating-point number.
complex(real[,imag])	Creates a complex number.
str(x)	Converts object x to a string representation.
repr(x)	Converts object x to an expression string.
eval(str)	Evaluates a string and returns an object.
tuple(s)	Converts s to a tuple.
list(s)	Converts s to a list.
set(s)	Converts s to a set.
dict(d)	Creates a dictionary. d must be a sequence of (key,value) tuples.
frozenset(s)	Converts s to a frozen set.
chr(x)	Converts an integer to a character.
unichr(x)	Converts an integer to a Unicode character.
ord(x)	Converts a single character to its integer value.
hex(x)	Converts an integer to a hexadecimal string
oct(x)	Converts an integer to an octal string.

Career Credentials

PYTHON-OPERATORS

BY

AMAR PANCHAL

Types of Operator

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Arithmetic Operators

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 31
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -11
* Multiplication	Multiplies values on either side of the operator	a * b = 210
/ Division	Divides left hand operand by right hand operand	b / a = 2.1
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 1
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity):	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

Comparison Operators

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a!= b) is true.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Assignment Operators

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Bitwise Operators

Operator	Description	Example
& Binary AND	Operator copies a bit, to the result, if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit, if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit, if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operand's value is moved left by the number of bits specified by the right operand.	a << = 240 (means 1111 0000)
>> Binary Right Shift	The left operand's value is moved right by the number of bits specified by the right operand.	a >> = 15 (means 0000 1111)

Logical Operators

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is False.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is True.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is True.

Membership Operators

- Python's membership operators test for membership in a sequence, such as strings, lists, or tuples.

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

Identity Operators

Identity operators compare the memory locations of two objects

Career Credentials

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is** **not**results in 1 if id(x) is not equal to id(y).

Operators Precedence

S.No.	Operator & Description
1	****** Exponentiation (raise to the power)
2	**~ + -** Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
3	*** / % //** Multiply, divide, modulo and floor division
4	**+ -** Addition and subtraction
5	**>> <<** Right and left bitwise shift
6	**&** Bitwise 'AND'

Operators Precedence

S.No.	Operator & Description
7	**^ |** Bitwise exclusive `OR' and regular `OR'
8	**<= < > >=** Comparison operators
9	**<> == !=** Equality operators
10	**= %= /= //= -= += *= **=** Assignment operators
11	**is is not** Identity operators
12	**in not in** Membership operators
13	**not or and** Logical operators

Career Credentials

PYTHON
CONDITIONAL STSTEMENT & LOOPS

BY

AMAR PANCHAL

IF

if Statements	It consists of a Boolean expression which results is either TRUE or FALSE followed by one or more statements.

```
if expression:
        #execute your code

if expression:
        #execute your code
else:
    #execute your code
```

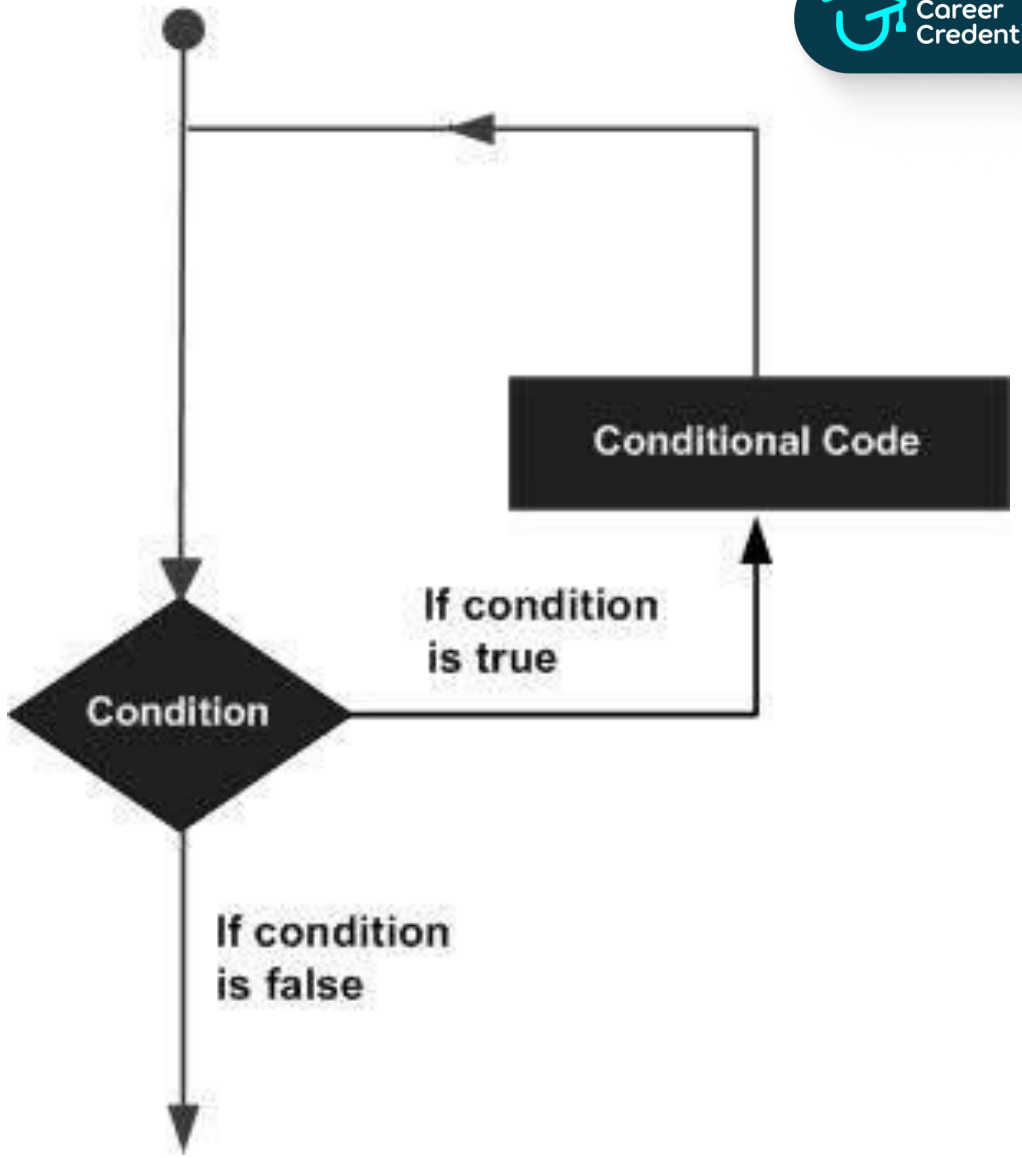
```
if expression:
        #execute your code
elif expression:
        #execute your code
else:
    #execute your code
```

Loops

•A loop statement allows us to execute a statement or group of statements multiple times.

•The following diagram illustrates a loop statement −

•



While

- Syntax

  while expression:

    statement(s)

Ex

```
count = 0
while (count < 5):
  print ('The count is:', count)
  count = count + 1

print ("Good bye!")
```

for Loop

- Syntax

for iterating_var in sequence:

  statements(s)

- The range() function

The built-in function range() is the right function to iterate over a sequence of numbers. It generates an iterator of arithmetic progressions.

Example

>>> range(5)

For Loop

- For i in range(5):

  print(i)

- For i in “python”:

  print(i)

Single Statement Suites

```
flag = 1
while (flag): print ('Given flag is really true!')
print ("Good bye!")
```

Loop Control Statements

S.No.	Control Statement & Description
1	**break statement:** Terminates the loop statement and transfers execution to the statement immediately following the loop.
2	**continue statement:** Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	**pass statement:** The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

else block with loops

```
no = int(input('any number: '))
numbers = [11,33,55,39,55,75,37,21,23,41,13]
    for num in numbers:
      if num == no:
            print ('number found in list')
            break
    else:
      print ('number not found in list')
```

```
numbers = [11,33,55,39,55,75,37,21,23,41,13]

for num in numbers:
   if num%2 == 0:
      print ('the list contains an even number')
      break
else:
   print ('the list does not contain even number')
```

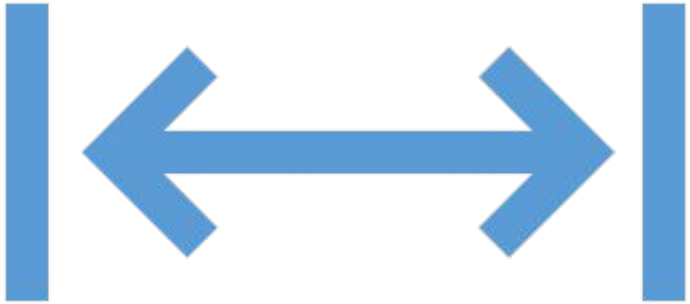
Career Credentials

Python-Functions

BY

AMAR PANCHAL

Defining a Function



- Function blocks begin with the keyword **def** followed by the function name and parentheses (( ))

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

Function Arguments

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Scope of Variables

- Global variables
- Local variables

```
total = 0

def sum( arg1, arg2 ):
   total = arg1 + arg2; # Here total is local variable.
   print ("Inside the function local total : ", total)
   return total

sum( 10, 20 )
print ("Outside the function global total : ", total )
```

Career Credentials

PYTHON –OOPS

BY AMAR PANCHAL

Creating Classes

```
class ClassName:
    'Optional class documentation string'
    class_suite
```

Creating Instance Objects

Accessing Attributes

- **The getattr(obj, name[, default]) – to access the attribute of object.**
- **The hasattr(obj,name) – to check if an attribute exists or not.**
- **The setattr(obj,name,value) – to set an attribute. If attribute does not exist, then it would be created.**
- **The delattr(obj, name) – to delete an attribute.**

Importing

Built-In Class Attributes

- **__dict__** – Dictionary containing the class's namespace.
- **__doc__** – Class documentation string or none, if undefined.
- **__name__** – Class name.
- **__module__** – Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **__bases__** – A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

Class Inheritance

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
   class_suite
```

check a relationships

- The **issubclass(sub, sup)** boolean function returns True, if the given subclass **sub** is indeed a subclass of the superclass **sup**.

- The **isinstance(obj, Class)** boolean function returns True, if *obj* is an instance of class *Class* or is an instance of a subclass of Class

Overriding Methods

S.No.	Method, Description & Sample Call
1	__init__ (self [,args...]) Constructor (with any optional arguments) Sample Call : obj = className(args)
2	__del__(self) Destructor, deletes an object Sample Call : del obj
3	__repr__(self) Evaluatable string representation Sample Call : repr(obj)
4	__str__(self) Printable string representation Sample Call : str(obj)
5	__cmp__ (self, x) Object comparison Sample Call : cmp(obj, x)

Data Hiding

- The __ sign

Career Credentials

PYTHON-EXCEPTIONS

BY

AMAR PANCHAL

EXCEPTION HANDLING

- **Exception & Assertions**

Handling an exception

try:

You do your operations here

.....

except ExceptionI:

If there is ExceptionI, then execute this block.

except ExceptionII:

If there is ExceptionII, then execute this block.

.....

else:

If there is no exception then execute this block.

The except Clause with Multiple Exceptions

try:

You do your operations here

.....

except(Exception1[, Exception2[,...ExceptionN]]]):

If there is any exception from the given exception list,

then execute this block.

.....

else:

If there is no exception then execute this block

The try-finally Clause

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

Argument of an Exception

```
try:
   You do your operations here
   ......................
except ExceptionType as Argument:
   You can print value of Argument here...
```

Raising an Exception

- Syntax

raise [Exception [, args [, traceback]]]

User-Defined Exceptions

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return self.value
try:
    raise MyError(2*2)
    except MyError, e:
    print("My exception occurred:", e.value)
```

Assertions in Python

- The syntax for assert is –

  assert Expression[, Arguments]

Career Credentials

PYTHON-FILES

BY

AMAR PANCHAL

The open Function

- file object = open(file_name [, access_mode][, buffering])

modes

MODE	
r	
r+	
w	
w+	
a	
a+	
rb	
wb	
ab	
rb+	
wb+	
ab+	

The file Object Attributes

S.No.	Attribute & Description
1	**file.closed** Returns true if file is closed, false otherwise.
2	**file.mode** Returns access mode with which file was opened.
3	**file.name** Returns name of the file.

code

```
f = open(“MyFile", “w")
print ("Name of the file: ", f.name)
print ("Closed or not : ", f.closed)
print ("Opening mode : ", f.mode)
f.close()
```

The close()

- fileObject.close()

Reading and Writing Files

- The **read()** method reads a string from an open file.
- fileObject.read([count]);

Reading and Writing Files

- The **write()** method writes any string to an open file.

```
fileObject.write(string);
f = open(“MyFile.txt", "w")
f.write( “file operation writing process”)
f.close()
```

File Positions

tell()	
seek(offset[, from])	

Renaming and Deleting Files

- Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files.To use this module, you need to import it.

rename()	**os.rename(current_file_name, new_file_name)**
remove()	os.remove(file_name)

Dealing With Directories

Command	Use
os.mkdir("newdir")	
os.chdir("newdir")	
os.getcwd()	
os.rmdir(“dirname”)	

Python Iterators, Generators, Decorators

BY

AMAR PANCHAL

What are iterators in Python?

- **Iterators are objects that can be iterated upon.**
- Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.
- Technically speaking, Python iterator object must implement two special methods, __iter__() and __next__(), collectively called the iterator protocol.

CODE

- my_list = [10, 20, 30, 40]
- my_iter = iter(my_list)
- print(next(my_iter))
- print(next(my_iter))
- print(my_iter.__next__())
- print(my_iter.__next__())

Building Your Own Iterator in Python

- We just have to implement the methods __iter__() and __next__().

CODE

```
class PowTwo:

        def __init__(self, max = 0):
                self.max = max
                self.n = 0

        def __next__(self):
                if self.n <= self.max:
                        result = 2 **
self.n
                        self.n += 1
                        return result
                else:
                    raise StopIteration
```

What are generators in Python?

- Python generators are a simple way of creating iterators.
- Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).
- It is fairly simple to create a generator in Python. It is as easy as defining a normal function with yield statement instead of a return statement.
- If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function. Both yield and return will return some value from a function.
- The difference is that, while a return statement terminates a function entirely, yield statement pauses the function saving all its states and later continues from there on successive calls.

Differences between Generator function and a Normal function

- Generator function contains one or more yield statement.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like __iter__() and __next__() are implemented automatically. So we can iterate through the items using next().
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, StopIteration is raised automatically on further calls.

CODE

```
# A simple generator functiondef
def my_gen():
        n = 1
        print('This is printed first')
       yield n
        n += 1
        print('This is printed second')
      yield n
      n += 1
      print('This is printed at last')
       yield n
```

String

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1,-1,-1):
```

Python Generator Expression

- my_list = [1, 3, 6, 10]
- a = (x**2 for x in my_list)

Why generators are used in Python?

1. Easy to Implement
2. Memory Efficient
3. Represent Infinite Stream

Python Decorators

- Python has an interesting feature called **decorators** to add functionality to an existing code.
- This is also called **metaprogramming** as a part of the program tries to modify another part of the program at compile time.

Functions are Objects

```
def first(msg):
        print(msg)
first("Hello")
second = first
second("Hello")
```

Functions inside Functions

```
def f():

    def g():
        print("Hi, it's me 'g'")#4
        print("Thanks for calling me")#5

    print("This is the function 'f'")#1
    print("I am calling 'g' now:")#2
    g()#3


f()
```

Functions as Parameters

```
def g():
    print("Hi, it's me 'g'")
    print("Thanks for calling me")

def f(func):
    print("Hi, it's me 'f'")
    print("I will call 'func' now")
    func()

f(g)
```

```
def g():
    print("Hi, it's me 'g'")
    print("Thanks for calling me")

def f(func):
    print("Hi, it's me 'f'")
    print("I will call 'func' now")
    func()
    print("func's real name is " + func.__name__)



f(g)
```

Functions returning Functions

```
def f(x):
    def g(y):
        return y + x + 3
    return g

nf1 = f(1)
nf2 = f(3)

print(nf1(1))
print(nf2(1))
```

A Simple Decorator

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper

def foo(x):
    print("Hi, foo has been called with " + str(x))

print("We call foo before decoration:")
foo("Hi")

print("We now decorate foo with f:")
foo = our_decorator(foo)

print("We call foo after decoration:")
foo(42)
```

The Usual Syntax for Decorators in Python

- The decoration in Python is usually not performed in the way we did it in our previous example, even though the notation foo = our_decorator(foo) is catchy and easy to grasp. This is the reason, why we used it! You can also see a design problem in our previous approach. "foo" existed in the same program in two versions, before decoration and after decoration.

- We will do a proper decoration now. The decoration occurrs in the line before the function header. The "@" is followed by the decorator function name.
- We will rewrite now our initial example. Instead of writing the statement
- foo = our_decorator(foo)
- we can write
- @our_decorator

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling: " + func.__name__)
        func(x)
        print("After calling :" + func.__name__)
    return function_wrapper

@our_decorator
def foo(x):
    print("Hi, foo has been called with " + str(x))

foo("Hi")
```

Regular expressions

The 14..

- \ Used to drop the special meaning of character following it
- [] Represent a character class
- ^ Matches the beginning
- $ Matches the end
- . Matches any character except newline
- ? Matches zero or one occurrence.
- | Means OR (Matches with any of the characters separated by it.
- * Any number of occurrences (including 0 occurrences)
- + One ore more occurrences
- {} Indicate number of occurrences of a preceding RE to match.
- () Enclose a group of REs

Function compile()

Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

```
# Module Regular Expression is imported using __import__().
import re

# compile() creates regular expression character class [a-e] which is equivalent to [abcde].
# class [abcde] will match with string with 'a', 'b', 'c', 'd', 'e'.
p = re.compile('[a-e]')

# findall() searches for the Regular Expression and return a list upon finding
print(p.findall("this is tony stark the iron man"))
```

Use of \

- Metacharacter blackslash ‘\’ has a very important role as it signals various sequences. If the blackslash is to be used without its special meaning as metacharacter, use’\\’
- \d Matches any decimal digit, this is equivalen to the set class [0-9].
- \D Matches any non-digit character.
- \s Matches any whitespace character.
- \S Matches any non-whitespace character
- \w Matches any alphanumeric character, this is equivalent to the class [a-zA-Z0-9_].
- \W Matches any non-alphanumeric character.

re.findall()

- Return all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found
  - import re
  - string = 'hi my main number is 9821 and other is 9892'
  - # A sample regular expression to find digits.
  - regex = '\d+'
  - match = re.findall(regex, string)
  - print(match)

```
import re
# \d is equivalent to [0-9].
p = re.compile('\d')
print(p.findall(“india was free on 15 aug 1947 at mid night 12.00am"))
```

split()

- Split string by the occurrences of a character or a pattern, upon finding that pattern, the remaining characters from the string are returned as part of the resulting list.
- Syntax :
  - re.split(pattern, string, maxsplit=0, flags=0)

  - from re import split
  - # '\W+' denotes Non-Alphanumeric Characters or group of characters
  - # Upon finding ',' or whitespace ' ', the split(), splits the string from that point
  - print(split('\W+', 'this are my final words'))

sub()

- The ‘sub’ in the function stands for SubString, a certain regular expression pattern is searched in the given string(3rd parameter), and upon finding the substring pattern is replaced by repl(2nd parameter), count checks and maintains the number of times this occurs.
- Syntax:
  - re.sub(pattern, repl, string, count=0, flags=0)

```
# Regular Expression pattern 'ub' matches the string at "Subject" and "Uber".
# As the CASE has been ignored, using Flag, 'uber' should match twice with the string
# Upon matching, 'uber' is replaced by 'OLA'
print(re.sub('uber', 'OLA' , 'Subject has Uber booked already', flags = re.IGNORECASE))
```

escape()

- Return string with all non-alphanumerics backslashed, this is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it.

re.search() :

- This method either returns None (if the pattern doesn’t match), or a re.MatchObject that contains information about the matching part of the string. This method stops after the first match, so this is best suited for testing a regular expression more than extracting data.

Django Framework

FRAMEWORK SETUP

1. RUN POWERSHELL IN ADMIN MODE
2. SAY: Set-ExecutionPolicy unrestricted-------say yes
3. Create dir for Django
4. In directory create virtual environment
    1. pip install virtualenv
5. Activate virtual environment
    1. virtualenv .
6. Activate scripts
    1. ./Scripts/activate
7. pip install django

Starting Project

1. Create folder for app
2. To create project
    1. django-admin.exe startproject <name>
3. Locate manage.py
4. Run server of Django
    1. python manage.py runserver
5. Check browser
    1. 127.0.0.0:8000 or localhost:8000
6. Can also handle migration error if any by
    1. Python manage.py migrate
7. Can see admin panel by
    1. Localhost:8000/admin/login

Creating super user

- Python manage.py createsuperuser
- Follow instruction on screen and remember username and password

Add module to your App

- Go to directory with manage.py
  - Python manage.py startapp <name>

Common Files Seen

- __init__.py
- admin.py
- apps.py
- models.py
- test.py
- views.py
- urls.py
- settings.py

To register app with base app

- Open settings.py from main app
  - To the list of Installed_app=[] add ‘<name of app>’,

Use urls.py

- Open and edit to make changes
  - See how route works in urlpatterns=[]

- For new sub app
  - Create urls.py in it local folder
  - Copy content of urls.py of base app
  - Remove admin statements

Add routes of sub app to main app urls.py

- Add
  1. From Django.urls import path,incude
  2. From <newapp> import views
  3. Add path(‘’,include(‘<subapp name>.urls’)),

Working with views.py

- In urls of subapp
  - from . imports views
- In views.py

  def home(request):

  return render(request,'home.html',{})

- Create templates folder in subapp
  - Create new file home.html
  - Code home .html
  - Save in templates folder only
  - In urls.py add
    - path('',views.home,name='home')

Working with templates

- Create a base file that is needed on every page
- Django creates base file and then extends it on every page

- Steps
  - Create base .html---code it
  - Add code blocks

    {% block <name> %}

    {% endblock %}
  - At the end and save

On other pages

- Add extends block

  {% extends ‘base.html’ %}

- Add block

  {% block <name> %}

  Page code

  {% endblock %}

For page title handling

- Create block title in title of base and then use it on every page
- <title>
- {% block title %}
- name the title
- {% endblock %}

Django links(dynamic)

- One can call pages by Django's url name given
- use

  <% url 'name of page' %>

Passing Paramenters

- See views .py which has a dictionary
- {key:value}
- One can define them and then call it directly or via data base
- Make changes in render of views.py

```
def home(request):
    name="amar"
    return render(request,"home.html",{'name':name})
```

Database handling

- Edit models.py in <new app>

- Create class that inherits (models.Model)
- Create all variables needed in the data base
  - Also code def __str__(self):
    Return self.<data>
- Use datatypes of Django
- From powershell
  - Python manage.py makemigrations
  - Python manage.py migrate

Data base in admin page

- Edit admin.py
- Add lines
- From .models import <classname>
- Admin.site.register(<classname>)

Adding database to page

- Edit views.py
- Add
  - From .models import <nameofdatabase>
  - Var=<database>.objects.all(if specific)

  At home page

  {‘key’:var}

  Use

  {%...%}

  For operations

Database creation and handling

- Steps
- 1 create class in models.py
- 2 create migration
- 3 push migration in database

In models.py

- class <classname>(models.Model):
- item=models.CharField(max_length=200)
- complete=models.BooleanField(default=False)

- def __str__(self):
- return self.item #what to return

migration

- Class DDL Database (automatically)
- Python manage.py makemigrations
- Python manage.py migrate

Register database in Admin section

- Use admin.py
- from .models import <class of models.py>
- admin.site.register(<class of models.py>)

To add database to page

- In views.py
- From .models import <name of class>
- To read all data
  - variable=<class>.objects.all
- At home():
  - Add {‘key’:var}
- On home.html add
  - {% for data in variable %}
    - {{data.items}}

Adding forms for input

- In forms.py(to be created)

```
from django import forms
from .models import Appdatabase

class AppdatabaseForm(forms.ModelForm):
 class Meta:
      model=Appdatabase
      fields=["item","complete"]
```

- In views.py add
  - from .forms import AppdatabaseForm

On base.html

- <form class="form-inline" method="POST">
- {%csrf_token%}
- <input class="form-control mr-sm-2" type="search" placeholder="Data to add" aria-label="" name="item">
- <button class="btn btn-outline-success my-2 my-sm-0" type="submit">Add to list</button>
- </form>

Views.py

- We need to add
  - from .forms import AppdatabaseForm

```
if request.method=="POST":
        form=AppdatabaseForm(request.POST or None)
        if form.is_valid():
            form.save()
            data=Appdatabase.objects.all
            return render (request, "home2.html" ,{'data':data})

    else:
        data=Appdatabase.objects.all
        return render (request, "home2.html" ,{'data':data})
```

Adding a prompt to a page

- Add
  - from django.contrib import messages
  - messages.success(request,("------>data added"))
- On home.html

```
{% if messages %}
    {% for message in messages %}
    <div class="alert alert-warning" role="alert">
        {{message}}
    </div>
    {% endfor %}
{% endif %}
```

Deleting from a form

- Add in urls.py
  - path("delete/<Appdatabase_id>",views.delete,name="delete"),
- In views.py

```
def delete(request,Appdatabase_id):
    item = Appdatabase.objects.get(pk=Appdatabase_id)
    item.delete()
    messages.success(request, ('Item Has Been Deleted!'))
    return redirect('home2')
```

On top of views.py

- from django.http import HttpResponseRedirect
- from django.shortcuts import render,redirect

- In home.html
- <td> Delete</td>