

- * Python is widely used general purpose, high level programming language.
- It was initially designed by Guido Van Rossum in 1991 & developed by Python Software Foundation.
- It was developed for emphasis on code readability, and its syntax allows programmers to express concepts in few lines of code.
- There are two major versions: Python 2 & Python 3.

* Beginning with Python Programming :-

- We need an interpreter to run our programs.
- * Online interpreters are also there.
- Windows :- There are many free interpreters available to run Python script like IDLE (Integrated Development Environment) which is installed when you install the Python software from <http://python.org>.
- Linux :- Python comes bundled with Linux.

- Python is not backward incompatible. Python 2.0 was released in 2000 and Python 3.0 was released in 2008. The most programs of earlier versions of Python could not be run using implementations of Python 3.0.

* Basic elements of Python :-

- A Python program sometimes called a SCRIPT, is a sequence of definitions and commands.
- These definitions are evaluated and commands are executed by the Python interpreter in something called SHELL.

Date _____
Page No. _____

* A new SHELL is created whenever execution of a program begins. Windows is associated with the shell.

→ A command, called a statement, instructs the interpreter to do something.

e.g. the statement `print "Hello"` instructs the interpreter to do something (output the string Hello) to the window associated with the shell.

* Objects, expressions and Numerical Types : *

→ Objects are the common things that Python programs manipulate. Every object has a type that defines the kinds of things that programs can do with objects of that type.

→ Two types of objects : Scalar & Non-Scalar

→ Scalar objects are indivisible, e.g. Think of them as the atoms of the language as atoms are indivisible.

→ Non-scalar objects, for example strings, have internal structure.

→ Python has four types of scalar objects :

1. `int` is used to represent integers. Literals of type `int` are written in the way we typically denote integers. (e.g. `-2` or `10` or `2000`)
2. `float` is used to represent real numbers. Literals of `float` always include a decimal point. (e.g. `3.0` or `3.14`)
3. `bool` is used to represent Boolean values `True` & `False`
4. `None` is a type with a single value.

→ Objects and Operators can be combined to form expressions, each of which evaluates to an object

of some type.

* We'll refer to this as the Value of the expression.
e.g. expression $3+2$ denotes the object 5 of type int and the expression $3.0+2.0$ denotes the object 5.0 of type float.

* $==$ op² is used to test whether two expressions evaluate to the same value.

$!=$ op² is used to test whether two expressions evaluate to different values.

* The symbol $>>>$ is a Shell Prompt indicating that the interpreter is expecting the user to type some Python code into the shell.

* The ^{Output} below the line with the prompt is produced when the interpreter evaluates the Python code entered at the prompt.

$>>> 3+2$	$>>> 3.0+2.0$	$>>> 3!=2$
5	5.0	True

* The built-in Python function `type` can be used to find out the type of an object:

$>>> \text{type}(3)$

O/P = `<type 'int'>` (Py 2.7)

O/P = `int` (Py 3)

* $\begin{pmatrix} i+j \\ \text{or} \\ i-j \end{pmatrix}$ = Sum of i and j or subtraction of j from i .
answer is int if both are int.
if either of them is float, result is float.

* Same for $i*j$.

* $i//j$ is the integer division.

e.g. value of $6//2$ is int 3.

$6//4$ is int 1.
The value is 1 because integer division returns the quotient & ignores the remainder.

* i/j is i divided by j .

Py-2.7 = When i & j are both of type int, the result is also an int.

$$1/2 = 0.5$$

Otherwise result is float.

Py-3.0 = '/' operator always returns a result float.

* $i \% j$: returns remainder. $i \bmod j$

$$\text{e.g. } 1 \% 2$$

Result is 1.

→ $i ** j$: i raised to power j . If i and j are both of type int, the result is an int. If either of them is float, the result is a float.

$$2 ** 3 \Rightarrow 8$$

→ The Comparison Operators are $==$, $!=$, $>$, $<$, $>=$, $<=$.

$$\text{e.g. } 2 >= 3 \\ \Rightarrow \text{false}$$

* Variables and Assignment :

→ Variables provide a way to associate names with objects.

$$\text{e.g. } \pi = 3$$

$$\text{radius} = 11$$

$$\text{area} = \pi * (\text{radius} ** 2)$$

So, it binds the name π and radius to different objects of type int. Then binds the name area to a third object of type int.

✓ → In Python, Variable names can contain uppercase and lower case letters, digits but they can't start with a digit, and a special character '_'.

→ Python names are case sensitive

e.g. June and june are different.

→ There are small numbers of reserved words (Keywords) in Python that have built-in meanings and can't be used as variable names.

for Python - 2.7 ⇒ and, as, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, with, while, yield.

→ for comments: we use #.

e.g. # Program for calculator
ans = a + b

→ Python allows multiple assignment.

e.g. x, y = 2, 3
binds x to 2 and y to 3.
x, y = 2, 3
print(x + y) ⇒ 5 //

* IDLE :→

Most programmers prefer to use some sort of text editor that is part of an integrated development environment (IDE).

→ We use IDLE, the IDE that comes as part of standard Python installation package.

→ IDLE is just an application. It provides,

→ A text editor with syntax highlighting, auto completion, smart indentation.

→ A shell with syntax highlighting.

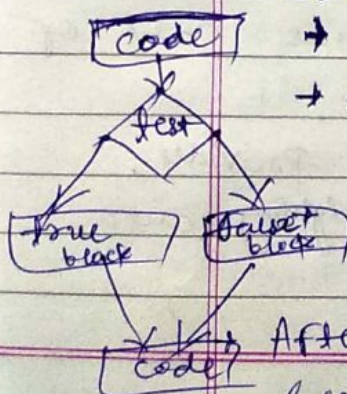
→ An integrated debugger.

- When IDLE starts it will open a shell window into which you can type python commands.
- It will also provide you with a file menu and edit menu.
- The file menu includes command to
 - Create a new editing window in which you can type a program.
 - Open a file containing an existing python program.
 - Save the contents of the current editing window into a file. (extension .py)
- The Edit menu includes - standard text editing commands (eg. copy, paste, find) plus some commands specifically designed to make it easy to edit python code. (eg. indent region and comment out region).

* Branching Programs :-

- The Straight line programs are those which execute one statement after another in the order in which they appear and stop when they run out of statements.

- Branching programs are more interesting. The simplest branching statement is Conditional. it has 3 parts
 - a test i.e. expression that evaluates to true or false.
 - a block of code that is executed if the test evaluates to true.



Optional block which executes if the statement is false.

After conditional statement is executed, execution resumes at the code following the statement.

Form :-

```
if Boolean Expression :
    block of code
else :
    block of code
```

e.g.

```
if x%2 == 0:
    print 'even'
else :
    Print 'odd' }
```

Py-2.7

e.g.

```
num = 0
if num >= 0:
    Print (" Positive or zero")
else:
    print (" Negative")
```

→ Indentation is Semantically meaningful in Python.

```
if x > 0:
    Print (" Positive")
Print ("Hello")
```

if above statement is indented, it will be a part of block of code associated with else. Otherwise block of code following the Conditional Statement.

→ Advantage of Python is that it ensures that the visual structure of a program is an accurate representation of the semantic structure of the program.

```
X = 10
if x%2 == 0:
    if x%3 == 0:
        Print ("divisible by 2 and 3")
    else:
        print ("divisible by 2 and not by 3")
elif x%3 == 0:
    print ("divisible by 3 and not by 2")
```


- Compound boolean expressions in the test of a Conditional, for example:

x = 10
y = 20
z = 5

if x < y and x < z:
 Print ("x is least")

elif y < z:
 Print ("y is least")

else:
 Print ("z is least")

- If a straight line program has n lines of code, it will take n units of time to run, whereas a branching program with n lines of code will take ^{less than} n units of time to run, but it can't take more, since each line of code is executed at most once.

* Strings and Input →

- Objects of type str are used to represent string of characters. Literals of type str can be written using either single or double quotes. e.g. 'abc' or "abc"

- The literal '123' denotes a string of characters not the number 123
- Print ("abc") Print ('abc')

- The operator '+' is overloaded. It has different meanings depending upon the types of the object to which it is applied. e.g. addition when applied to two numbers and Concatenation when

applied to two strings.

→ The Operator '*' is also overloaded. It means what you expect it to mean when its operands are both numbers.

→ When applied to an int and a str, it duplicates the str.

e.g.

`print(2 * 'abc')`
⇒ `abcabc`

`print(2 * 3)`
⇒ `6`

e.g. `print(a)`
⇒ error

`print('a' * 'a')`
⇒ error (can't multiply)

Interpreter treats it as a name which is not bounded to any object, so error.

→ Type Checking is not strong as in JAVA.

→ `'4' < 3` (type error `str() < int()`)

The designers of Python decided that it should be false, because all numeric values should be less than all values of type str.

→ Strings share the following operations with all the sequence types.

→ The length of a string can be found using `len` function.

e.g. `print(len('abc'))` ⇒ `3`

→ Indexing can be used to extract individual characters from a string.

→ In python all indexing is Zero-based.

`print('abc'[0])` will print `a`.

`print('abc'[3])` will generate index out of range error.

→ Negative numbers are used to index from the end of a string.

e.g. `Print('abc'[-1])`
⇒ C

* Slicing is used to extract substrings of arbitrary length. If `s` is a string, the expression `s[start:end]` denotes the substring of `s` that starts at index start and ends at index end-1.

e.g. `Print('abc'[1:3])` ⇒ bc $(1 \leq \dots < 3)$
`Print('abcde'[1:4])` ⇒ bcd $(1 \leq \dots < 4)$
`Print('abcde'[1: len('abcde')])`
 $1: 5$
So, from 1 to $5-1=4$
 ⇒ bcde

* Input :-

user can get i/p directly. There are two functions

↳ input and raw_input.

→ each takes a string as an argument and displays it as a prompt in the shell.

→ it then waits for the user to type something, followed by hitting the enter key.

→ raw_input :- the input line is treated as a string and it becomes the value returned by the

→ input :- it treats the typed line as a python expression and infers a type.

e.g. `name = raw_input('enter name')`

⇒ enter name : xyz

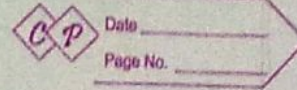
`Print('name is', name)`

⇒ name is xyz

`Print('name is' + name + '?')`

⇒ name is xyz ?

Python 3 has only one Command : Input



e.g.

```
name = input('enter name:')  
print('name is ' + name)  
print(type(name))
```

$\Rightarrow$	enter name : MANISHA name is MANISHA <Class 'str'>		enter name : 3 name is 3 <Class 'string'>
---------------	--	--	---



Var is bound to string not to the integers.

* Type Casting (Type Conversion) :

<pre>print(int(3.5))</pre>		<pre>print(float(5))</pre>
$\Rightarrow 3$		$\Rightarrow 5.0$

* Iteration :
(Looping)

If the test evaluates to true, the program executes the loop body once, and then goes back to reevaluate the test. This process is repeated until the test evaluates to False.

e.g.

```
x = 3  
ans = 0  
Count = 1  
i = x  
while (i != 0):  
    ans = ans + x  
    i = i - 1
```

O/P:
 $3 * 1 = 3$
 $3 * 2 = 6$
 $3 * 3 = 9$

```
print(str(x) + '*' + str(Count) + '=' + str(ans))  
Count = Count + 1
```


* For loop :-

for Variable in Sequence :
Code block

The Variable : bound to the first value in the Sequence

The process is executed until the sequence is exhausted or a break statement is executed within the code block.

→ The sequence of values bound to variable is most commonly generated using built-in function range, which returns a sequence containing an arithmetic progression.

e.g.

$\text{range}(5, 40, 70)$ produces the sequence $[5, 15, 25, 35, 45, 55, 65, 75]$ (Note: 70 is not included)
 $\text{range}(40, 5, 70)$ produces the sequence $[40, 30, 20, 10]$ (Note: 5 is not included)

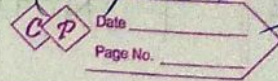
if the first argument is omitted it defaults to 0 and if the last argument is omitted it defaults to 0.
 e.g. $\text{range}(0, 3)$ and $\text{range}(3)$ both produce the sequence $[0, 1, 2]$.

<pre> x = 4 for i in range(0, x): print i </pre>	<pre> O/P = 0 1 2 3 </pre>
--	----------------------------

<pre> x = 4 for i in range(0, x): print(i) </pre>	<pre> O/P = 0 1 2 3 </pre>
---	----------------------------

~~xc = 5~~

for a single line: `Print(x, end=" ")`



It raises the question, whether the changing the value of x inside the loop affects the number of iterations. It does not affect the arguments to the range function in the line with `for` are evaluated just before the first iteration of the loop and not reevaluated for subsequent iterations.

<code>x = 4</code>	0
<code>for j in range(x):</code>	1
<code>for i in range(x):</code>	2
<code>print(i)</code>	3
	4
	5
<code>x = 2</code>	6

because the range function in the outer loop is evaluated only once, but the range function in the inner loop is evaluated each time the inner `for` statement is reached.

reason: range is not dynamic. it creates a list, and iterates through it. `range(x)` will create list of 0 until x . Value change in x does not impact that loop. now, the loop inside will change but not the outside one.

`x = 4`
`for j in range(x):` → `for j in [0, 1, 2, 3]`
 `for i in range(x):` → `for i in [0, 1, 2, 3]`
 `print(i)`

`x = 2`

`for i in range(x):` → `for i in [0, 1]`

`for i in range(x)` → `for i in [0, 1]`

`for i in range(x)` → `for i in [0, 1]`

* Functions & Scoping :-→ Function definitions :-form :

```
def name of function (list of formal parameters)
    body of function
```

e.g.

```
def maxval(x, y):
    if x > y:
        return x
    else:
        return y
```

→ def is reserved, maxval is fn name, x, y in parenthesis are formal parameters of the function. When the function is used, the formal parameters are bound to the actual parameters (often referred to as arguments) of the fn invocation (fn call).

e.g. maxval(3, 4) ⇒ 4.
binds x to 3 and y to 4.

→ The fn body is a piece of python code. return can be used only within the body of a fn.

* A function call is an expression, which has a value. That value is the value returned by the invoked function.

e.g. maxval(3, 4) + maxval(3, 2)
 $\downarrow \quad \quad \downarrow$
 4 + 3
 = 7

→ Parameters provide something called Lambda Abstraction, allowing programmers to write the code that manipulates not specific objects, but instead whatever objects the caller of the fn chooses to use.

* Keyword Arguments and Default Values :-

- In python there are two ways that formal parameters get bound to actual parameters.
- The first method - Positional - the first formal parameter is bound to first actual parameter. The second formal to second actual.
- python also supports keyword arguments, in which formals are bound to actual using the name of the formal parameters.

e.g.

```
def printname(firstname, lastname, reverse):
    if reverse:
        print(lastname + ', ' + firstname)
    else:
        print(firstname, lastname)
```

This function assumes firstname and lastname as strings and reverse as boolean.

if reverse == True it prints lastname, firstname otherwise firstname, lastname.

- each of the following is an equivalent invocation of printname:

```
printname('ab', 'cd', False)
printname('ab', 'cd', reverse=False)
printname('ab', lastname='cd', reverse=False)
printname(lastname='ab', firstname='cd', reverse=False)
```

- Though the keyword arguments can appear in any order in the list of actual parameters, it is not legal to follow a keyword argument with a non-keyword argument.

So, error message will be produced by -
`printname('ab', lastname='cd', False)`

→ Keyword Arguments are commonly used ~~to~~ in conjunction with default parameter values.
e.g.

```
def printname (firstname, lastname, reverse=False)
    if reverse:
        print (lastname + ', ' + firstname)
    else:
        print (firstname, lastname)
```

default values allow programmers to call a fn with fewer than the specified number of arguments.

e.g.

```
printname ('ab', 'cd')
printname ('ab', 'cd', True)
printname ('ab', 'cd', reverse=True)
```

o/p: cd, ab
 ab, cd
 ab, cd.

*

Scoping :-

```
def f(x): # name x is used as formal parameter
    y=1
    x=x+y
    print ("x=", x)
    return x.
```

o/p:

x=4

z=4

x=3

y=2

x=3

y=2

z=f(x) # value of x used as actual parameter

print ("z=", z)

print ("x=", x)

print ("y=", y)

→ At the call of f , the formal parameter x is locally bound to the value of the actual parameter x .

↳ Each f^n defines a new namespace, also called a scope. The formal parameter x and local variable y that are used in f exist only within the scope of the definition of f .

$x = x + y$ binds x to f .

↳ The assignment in f (function) have no effect at all on the bindings of the name x & y that exist outside the scope of f .

→ At top level, e.g. the level of the shell, a symbol table keeps track of all names defined at that level and their current bindings.

→ When a function is called, a new symbol table (stack frame) is created. This table keeps track of all names defined within the function (including formal parameters) & their current bindings. If a function is called from within the f^n body, yet another stack frame is created.

→ When the function completes, its stack frame goes away.

→ In Python, one can always decide the scope of a name by looking at a program text. This is called static or lexical scoping.


```

def f(x):
    def g():
        x = 'abc'
        print('x=', x)
    def h():
        z = x
        print('z=', z)
    x = x + 1
    print('x=', x)
    h()
    g()
    print('x=', x)
    return g

```

```

x = 3
10 1 z = f(x)
11 11 print('x=', x)
12 12 print('z=', z)
13 13 z()

```

O/P:

```

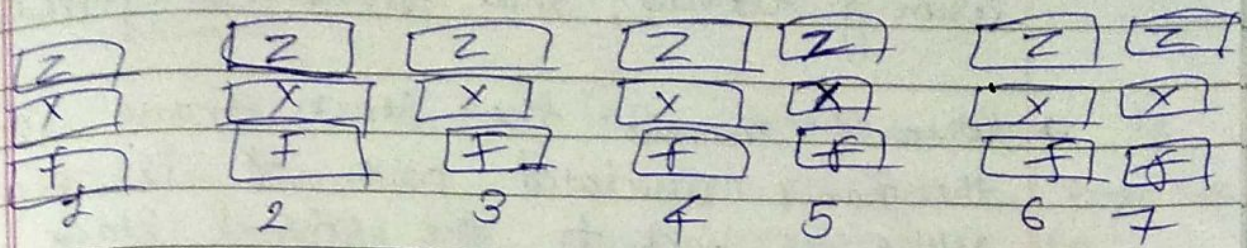
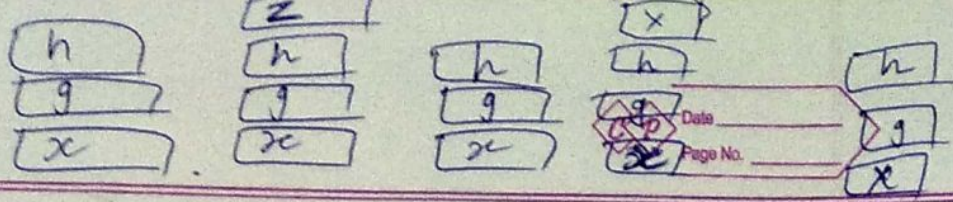
// 4
3 print x=4
5 print z=4
7 x=abc
8 x=4
11 x=3
12 z = <function
13 f(locals), g at 007f0
    X=abc

```

→ The history of the stack frames associated with the code. The first column contains the set of names known outside the body of the function f . i.e., the variables x and z and function name f .

→ The first assignment statement binds x to 3. The assignment $z = f(x)$ first evaluates the expression $f(x)$ by invoking f^n with the value to which x is bound.

→ When f is entered, a stack frame is created as shown in column 2. The names in the stack frame are x, g, h . The variable g and h are bound to objects of type function. The properties of these f^n are given by $f^n \text{ def}^n$ within f .



- When h is invoked from within f , yet another stack frame is created, in column 3. This frame only contains local variable z .
- ↳ it does not contain x why? a name is added to the scope associated with the function body only if that name is either a formal parameter or variable that is bound to an object within the body of the function. In the body of h , x occurs only on the right handside of an assignment statement.
- ↳ The appearance of name (x) that is not bound to an object anywhere in the f^n body (h) causes the interpreter to search the stack frame associated with the scope within which the f^n is defined (f^m f). if the name is found, the value ~~to~~ which it is bound (4) is used. If it is not found, the error message is produced.
- When h returns, the stack frame associated with the invocation of h goes away. (Popped off) as given in column 4.
- ↳ We never remove frames from the middle of the stack, but only one recently added frame because LIFO for the stack.
- Next g is invoked, and a stack frame containing g 's local variable x is added. (column 5).

When g returns, that frame is popped (Columns)

- When f returns the stack frame containing the names associated with f is popped, getting us back to the original stack frame.
- When f returns, even though the variable g no longer exists, the object of type function to which that name was once bound still exists, because, f^n are objects and can be returned like any other object.
- So, z can be bound to the value returned by f and f^n call $z()$ can be used to invoke the f^n that was bound to the name g within f - even though the name g is not known outside the context of f .

ex

```
def g():
    print(x)
    x=2
```

```
x=3
g()
```

error message:

UnboundLocalError: local variable 'x' referenced before assignment

- it happens because, the $x=2$ statement causes x to be local to $f^n g$. So, it has no value when ~~the~~ print statement is executed.

* Specifications :-

→ The text between the Triple Quotation marks is called a docstring in a Python. Python programmers use a docstring to provide specifications of functions. These docstring fⁿ can be accessed using the built-in fⁿ help.

eg. If we enter the shell and type
 // help(abs), system will display
 abs(x)

returns absolute value of the argument.

eg.

```
def root(x, power, epsilon)
    """ Assumes x and epsilon as int or float """
```

→ help(root) will display above specification.

↳ A specification of a fⁿ defines a contract between the implementer of a fⁿ and those who will be writing programs that use the function.

→ we'll refer the user of fⁿ as clients.

This contract can be thought of as containing two parts:

Assumptions: These describe conditions that must be met by clients of the fⁿ. (Constraints on actual parameters)

Guarantees: These describe conditions that must be met by the fⁿ, provided that it has been called in a way that satisfies the assumptions.

* Decomposition :- Creates structure It allows us to break a program into parts that are reasonably self-contained and that may be reused in different settings.

- Abstraction: It hides detail. It allows us to see a piece of code as if it were a black box — that is interior, whose details we can't see.
- It means preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context.
- Specification of any f^n is an abstraction of all the possible implementations that meet the specification.

* Recursion :-

* Iterative and Recursive Implementations of factorial

```
def fact(n):
    """ Assumes n an int > 0
        returns n! """
    result = 1
    while n > 1:
        result = result * n
        n -= 1
    return result
```

```
def fact(n):
    """ Assumes n an int > 0
        returns n! """
    if n == 1:
        return n
    else:
        return n * fact(n-1)
```

* Global Variables :-

```
def fib(x):
    """ Assumes x >= 0, returns fibo(x) """
    global num
    num += 1
    if x == 0 or x == 1:
        return 1
    else:
        return fib(x-1) + fib(x-2)
```



```
def testfib(n):
    for i in range(n+1):
```

global num

num = 0

print('fib of', i, '=', fib(i))

print('fib called', num, 'times')

O/P:

fib of 0 = 1

fib called 1 times

fib of 1 = 1

fib called 1 times

fib of 2 = 2

fib called 3 times

fib of 3 = 3

fib called 5 times

fib of 4 = 5

fib called 9 times

fib of 5 = 8

fib called 15 times

fib of 6 = 13

fib called 25 times

Explanation

i	num	Fib(i)	num
0	0	1	1
1	0	1	1
2	0	Fib(1)+fib(0)	1
		↓ ↓	2
		1 + 1 = 2	3

3	0	Fib(2)+fib(1)	1	=>	fib(2)+fib(1)
		2 + 1			num=2
		= 3			fib(2)=fib(1)+fib(0)
					num=3
					1 + fib(0)
					num=4
					1 + 1 = 2
					num=5
					2 + fib(1)

* Modules :

As programs get larger it is more convenient to store different parts in different files.

eg.

if multiple people are working on same program it would be nightmare if all try to update the same file.

→ Python allows to easily construct a program from code in multiple files.

* A module is a .py file containing python definitions and statements.

→ we create circle.py containing the code:

```
Pi = 3.14
def area(radius):
    return Pi * (radius ** 2)
def circumference(radius):
    return 2 * Pi * radius
```

→ A program gets access to a module through an import statement.

```
import circle
Pi = 3
print(Pi)
print(circle.pi)
print(circle.area(3))
print(circle.circumference(3))
```

→ 3
→ 3.14
→ 28.27
→ 18.84

Modules are typically stored in individual files. Each module has its own ^{private} symbol table.

→ `import M` creates binding for module M in the scope in which the import appears.

CP Date _____
Page No. _____

in the importing context we use dot notation to indicate that we are referring to a name defined in the imported module.

→ Module is imported only once per interpreter session. If you startup a shell, import a module, and then change the contents of that module, the interpreter will still be using the original version of the module. This can lead to puzzling behaviour when debugging. So, start a new shell.

* Files :-

Every computer system uses files to save things from one computation to the next. Python provides many facilities for creating and accessing files.

→ Python achieves OS independence by accessing files through something called a file handle.

```
nameHandle = open('kids', 'w')
```

instructs the operating system to create a file with name kids and return a file handle for that file.

→ 'w' indicates file is opened for writing.

it is open to close the file when the program is finished using it.

e.g.

```
nameHandle = open('kids', 'w')
```

```
for i in range(2):
```

```
    name = input('enter name:')
```

```
    nameHandle.write(name + '\n')
```

```
nameHandle.close()
```

This code opens a file and write two lines & closes the file.

→ We can open the file for reading using argument 'r'.

```
nameHandle = open('kids', 'r')
for line in nameHandle:
    print(line)
nameHandle.close()
```

Since python treats a file as a sequence of lines, we can use a for statement to iterate over the file's contents.

→

```
nameHandle = open('ff', 'r')
for line in nameHandle:
    print(line[:-1])
```

 ⇒ from 0 to end of file

```
nameHandle.close()
```

→ if we want to add new extra ~~text~~ in a file, we can append the content by using argument 'a'.

```
nameHandle = open('kids', 'a')
nameHandle.write('abn')
nameHandle.write('cdin')
nameHandle.close()
```

```
nameHandle = open('kids', 'r')
for line in nameHandle:
    print(line[:-1])
nameHandle.close()
```

it will print

ab
cd

★ Operation on file :

`Open(fn, 'w')` : fn is a string representing a filename. Creates a file for writing and returns a file handle.

`open(fn, 'r')`: Opens existing file for reading & returns a file handle.

`open(fn, 'a')`: Opens an existing file for appending and returns a file handle.

`fn.read()`: Returns the content associated string of the file handle `fn`.

`fn.readline()`: Returns the next line associated with filehandle `fn`.

`fn.readlines()`: Returns a list each element of which is one line of the file associated with file handle `fn`.

`fn.write(s)`: Writes the string `s` to the end of the file associated with the file handle `fn`.

`fn.writelines(s)`: `s` is a sequence of strings. Writes each element of `s` as a separate line to the file associated with file handle `fn`.

`fn.close()`: Closes the file associated with `fn`.

* Tuples :-

(unchangeable)

like strings, tuples are immutable ordered sequence of elements.

→ The difference between strings and tuples is that the elements of tuple need not be characters.

→ The individual elements can be of any type, & need not be of the same type as each other.

→ Literals of type tuple are written by enclosing a comma-separated list of elements within parenthesis.

e.g. `t1 = ()` // null tuple.

`t2 = (1, 'two', 3)`

`print(t1)`

`print(t2)`

⇒ O/p: `()`

`(1, 'two', 3)`

e.g. `print(3 * ('a', 2))`

O/p ⇒ `('a', 2, 'a', 2, 'a', 2)`

→ Like strings, tuple can be concatenated, indexed and sliced.

`t1 = (1, 'two', 3)`

`t2 = (t1, 3.25)`

`print(t2)`

`print((t1 + t2))`

`print((t1 + t2)[3])`

`print((t1 + t2)[2:5])`

⇒ `((1, 'two', 3), 3.25)`

⇒ `(1, 'two', 3, (1, 'two', 3), 3.25)`

⇒ `(1, 'two', 3)`

⇒ `(3, (1, 'two', 3), 3.25)`

Ex. a function which returns common elements that are in tuple t_1 and t_2 .

```
def intersect (t1, t2):
    result = () # initially null tuple
    for e in t1:
        if e in t2:
            result += (e,)
    return result
```

* Sequence and multiple assignments

$x, y = (3, 4)$

So, x will be bound to 3 and y to 4.

$a, b, c = 'xyz'$

so, a will bind to 'x', b will bind to 'y' and c will bind to 'z'.

* ~~Ques~~ ~~and~~ ~~Ans~~ Ranges :-

→ like strings and tuples, ranges are immutable.

→ The range function returns an object of type range.

→ range f^n takes 3 arguments: Start, stop, step and returns the progression of integers Start, Start + step, Start + 2 * step etc.

Ex. If only two arguments are supplied, a step of 1 is used.

If only one argument is supplied, that the argument is stop, Start defaults to 0 and step defaults to 1.

✓ All of the operations on tuples are also available for ranges, except for Concatenation and Repetition.

e.g. range(10)[2:6][2]

So, range is (10) $\Rightarrow$ Start=0, Stop=10, Step=1

O/P: 0 1 2 3 4 5 6 7 8 9

now, [2:6] $\Rightarrow$ 2 3 4 5

from 2 to 5 step(index)

(Slicing)

now, [2] $\Rightarrow$ 2 3 (4) 5
[0] [1] [2] [3]

So, answer is 4.

→ When the == operator is used to compare objects of type range, it returns True if the two ranges represent the same sequence of integers.

e.g. range(0, 7, 2) == range(0, 8, 2) evaluates to True.

0, 2, 4, 6 = 0, 2, 4, 6, So True

e.g. range(0, 7, 2) == range(6, -1, -2) evaluates to False

0, 2, 4, 6 $\neq$ 6, 4, 2, 0, False because different order

→ Unlike objects of type tuple, the amount of space occupied by an object of type range is not proportional to its length, because a range is freely defined by start, stop and step, which can be stored in small amount of space.

→ Mostly Range is used in for loops but, objects of type Range can be used anywhere, a sequence of integers can be used.

* Lists and Mutability :->

like a tuple, a List is an ordered sequence of values, where each is identified by an Index.

→ The Syntax is same as tuple except Square brackets.

→ The empty list is written as [].

```
L = ['I did it all', 4, 'bcy']
for i in range(len(L)):
    print(L[i])
```

O/p:

I did it all
4
bcy

Ex:

[1, 2, 3, 4] [1:3] [1]

so [2, 3]
now [3] so Answer is 3.

→ Difference between Tuple and List:

Lists are Mutable (changeable) where as tuples and strings are Immutable (unchangeable).

→ There are many op^r that can be used to create objects of these immutable types and Variables can be bound to objects of these types. But objects of immutable types can't be modified.

→ Objects of type List can be modified after they are created.

→ The distinction betⁿ mutating an object and assigning an object to a variable may, at first appear subtle.

When the statements

Techs = ['MIT', 'CAL']

Ivys = ['Harvard', 'Yale', 'Brown']

are executed, the interpreter creates two new lists and binds appropriate variables to them.

The assignment statements,

u1 = [Techs, Ivys]

u2 = [['MIT', 'CAL'], ['Harvard', 'Yale', 'Brown']]

also creates new lists and binds variables to them. The elements of these lists are themselves lists.

print(u1 == u2)

print(u1 == u2)

print(u1 == u2)

Produces the output

u1 = [['MIT', 'CAL'], ['Harvard', 'Yale', 'Brown']]

u2 = [['MIT', 'CAL'], ['Harvard', 'Yale', 'Brown']]

True.

→ It appears as if u1 and u2 are bound to the same value. But appearance can be deceiving. As u1 and u2 are bound to quite different values.

→ The u1 and u2 are bound to different objects. Can be verified using the built-in python function id, which returns a unique integer identifier for an object.

id - This fⁿ allows us to test for Object Equality.


```
Print (u1 == u2) # test Value equality
Print (id(u1) == id(u2)) # test object equality
Print ("Id of u1 =", id(u1))
Print ("Id of u2 =", id(u2))
```

O/P:

True

False

Id of u1 = 4447805768
Id of u2 = 4456134408 } This can be different when we run the code

```
* Print ("Ids of u1[0] and u1[1]", id(u1[0]), id(u1[1]))
Print ("Ids of u2[0] and u2[1]", id(u2[0]), id(u2[1]))
which prints,
```

Ids of u1[0] and u1[1] 4447807688 4456134684

Ids of u2[0] and u2[1] 4447805768 447806728

It appends code at the end.

Techs.append('RPI')

it appends new element to list Techs.

Techs = ['MIT', 'CAL', 'RPI']

The append Method has a side effect. Rather than create a new list, it mutates the existing list Techs by adding a new element 'RPI' to the end of it.

now, print (u1 = u1)
print (u2 = u2)

Will produce the output

u1 = ['MIT', 'CAL', 'RPI'], ['Harvard', 'Yale', 'Brown']
u2 = ['MIT', 'CAL'], ['Harvard', 'Yale', 'Brown']

→ This is called ALIASING. There are two distinct paths to the same list object. One path is through the variable Techs and other

through the first element of the list object to which it is bound.

* When we append one list to another e.g., Tech.append(Ivys), the original structure is maintained.

+ suppose we don't want to maintain this structure, but want to add the elements of one list into another list. We can do that by using list concatenation or the extend method.

e.g.

L1 = [1, 2, 3]

L2 = [4, 5, 6]

L3 = L1 + L2

Print('L3 = ', L3)

L1.extend(L2)

Print('L1 = ', L1)

L1.append(L2)

print('L1 = ', L1)

O/p:

L3 = [1, 2, 3, 4, 5, 6]

L1 = [1, 2, 3, 4, 5, 6]

L1 = [1, 2, 3, 4, 5, 6, [4, 5, 6]]

Operator '+' creates a new list and returns it. In contrast, extend and append both mutated L1.

Methods Associated with List

* L.append(e) - adds the object e to the end of L.

L.count(e) - returns the number of times that e occurs in L.

- $L.insert(i, e)$ - inserts the object e into L at index i .
- $L.extend(L2)$ - Adds the item in List $L2$ to the end of L .
- $L.remove(e)$ - delete the first occurrence of e from L .
- $L.index(e)$ - returns the index of the first occurrence of e in L , raises an exception if e is not in L .
- $L.pop(i)$ - removes & returns the item at index i in L , raises an exception if L is empty. If i is omitted, it defaults to -1 , to remove & return the last element of L .
- $L.sort()$ - sorts the elements of L in ascending order.
- $L.reverse()$ - reverses the order of the elements in L .

* CLONING :-

→ it is good to avoid mutating (changing) a list over which one is iterating.

e.g.

```
def remove(L1, L2)
```

""" removes elements from $L1$ which are also in $L2$ """

```
for e1 in L1:
```

```
    if e1 in L2:
```

```
        L1.remove(e1)
```

```
L1 = [1, 2, 3, 4]
```

```
L2 = [1, 2, 5, 6]
```

```
remove(L1, L2)
```

```
Print('L1 =', L1)
```

O/P: [2, 3, 4]

here the problem is that 2 is duplicate, still it will remain in list.

→ If the list is mutated, the hidden Counter starts at 0, discovers that $l[0]$ is in l_2 , and removes it - reducing the length of l_2 to 3. The Counter is then incremented to 1, and the code proceeds to check if the value of $l[1]$ is in l_2 .

+ Notice that this is not the original value of $l[1]$ (i.e., 2) but rather the current value of $l[1]$ (i.e., 3). As you can see, it is possible to figure out what happens when the list is modified within the loop.

→ So, to avoid this kind of problem is to use Slicing to clone (i.e. making a copy of) the list and write for l_2 in $l[:]$.

note that

$newl = l$

for l_2 in $newl$:

could not solve the problem.

It would not create a copy but would create a new ~~new~~ name for the existing list.

→ Slicing is ~~the~~ not ^{the} only way to clone lists in python. The expression $list(L)$ returns a copy of the list L . If the list to be copied contains mutable objects that you want to copy as well, import the standard library module copy and use $copy.deepcopy$.

* List Comprehension :->

- It provides a way to apply an operation to the values in a sequence.
- It creates a new list in which element is the result of applying a given operation to a value from a sequence.

eg,

```
L = [x**2 for x in range(1,7)]
print(L)
```

will print,

```
[1, 4, 9, 16, 25, 36]
```

* Functions as Objects :->

- Functions are first class objects. It means they can be treated like objects of any other type. eg., int or list.
- using fⁿ as arguments allows a style of coding called Higher Order Programming.

eg,

```
def applytoeach(L, f):
    """ Assumes L is a list, f is a function.
    Mutates L by replacing each element, e, of L by f(e) """
    for i in range(len(L)):
        L[i] = f(L[i])

def factR(n):
    if n==0:
        return 1
    else:
        return n * factR(n-1)

L = [1, -2, 3.33]
print('L =', L)
print('Apply abs to each element of L')
applytoeach(L, abs)
print('L =', L)
```



```

Print('Apply int to each element of', L)
applytoeach(L, int)
Print('L =', L)
Print('Apply factorial to each element of', L)
applytoeach(L, fact)
Print('L =', L)
Print('Apply fibonacci to each element of', L)
applytoeach(L, fib)
Print('L =', L)

```

O/P:

```

L = [1, -2, 3.33]
Apply abs to each element of L
L = [1, 2, 3.33]
Apply int to each element of [1, 2, 3.33]
L = [1, 2, 3]
Apply factorial to each element of [1, 2, 3]
L = [1, 2, 6]
Apply fibonacci to each element of [1, 2, 6]
L = [0, 1, 0, 1, 1, 0, 1, 1, 2] [1, 2, 13]

```

* When used in a for loop, map behaves like range function in that it returns one value for each iteration of the loop. These values are generated by applying the first argument to each element of the second argument.

e.g., for i in map(fib, [2, 6, 4]):
Print(i)

O/P: 2
13
5

→ Map does not return a value at a time but it returns a list of values.

map can be fⁿ of n arguments.

C.P.

Date

Page No.

e.g. L1 = [1, 28, 50]
L2 = [2, 57, 9]
for i in map(min, L1, L2):
 print(i)

O/P:

1
28
9

* Python supports creation of Anonymous functions (i.e., that are not bound to a name), using a reserved word lambda.

The general form is:

lambda <sequence of variable names>: <expression>

e.g., lambda x, y: x*y returns a function that returns a product of its two arguments. Lambda expressions are frequently used as arguments to the higher order function.

e.g., L = []
for i in map(lambda x, y: x**y, [1, 2, 3, 4], [2, 3, 4]):
 L.append(i)

O/P: print(L)
[1, 4, 3, 1] (returns 1², 2³, 3⁴, 4⁰)

→ it works like inline fⁿ.

* Strings, Tuples, Ranges, Lists →

(Above four)

+ Common Operations on Sequence types :-

seq[i] returns the ith element in the sequence.

len(seq) returns the length of the sequence

seq1 + seq2 returns the Concatenation of the two sequences
(not available for range)

$n * seq$ - returns a sequence that repeats sequence n times. (not available for range)

$seq[start:end]$ - returns a slice of the sequence.

$e \text{ in } seq$ - is true if e is contained in the sequence and false otherwise.

$e \text{ not in } seq$ - is true if e is not in the sequence and false otherwise.

for $e \text{ in } seq$ - iterates over the elements of the sequence.

Sequence Type	Type of elements	Example of literals	Mutable
Str	Characters	'a', 'abc'	No
Tuple	any Type	(), (3,), ('abc', 4)	No
range	Integer	range(10), range(1, 10, 2)	No
List	any Type	[], [3], ['abc', 4]	Yes

Python Programmers tend to use lists far more than tuples, because Lists are Mutable. They can be constructed incrementally during a computation.

e.g. following code constructs a list containing all of the even numbers in another list.

```
evenelem = []
for e in L:
    if e % 2 == 0:
        evenelem.append(e)
```

one Advantage of Tuple is that as they are immutable, Aliasing is never a worry. another advantage of their being immutable is

that tuples, unlike lists, can be used as keys in dictionaries.

- Since strings only contain characters, they are considerably less versatile than tuples or lists.
- on the other hands when you work with strings of characters there are many inbuilt methods that make life easy!

→ e.g.,

one of the more useful built-in methods is split, which takes two strings as arguments. The second argument specifies a separator that is used to split the first argument into a sequence of substrings.

e.g.

```
print('hi hello hey-hiii-he'.split(' '))
print('hi hello hey-hiii-he'.split('-'))
print('hi hello hey-hiii-he'.split('--'))
```

O/P:

```
['hi', 'hello', 'hey-hiii-he']
['hi hello hey', 'hiii', 'he']
['hi hello hey-hiii', 'he']
```

* Some Methods on Strings :

S.count(st) - Counts how many times string st occurs in S.

S.find(st) - returns the ~~first~~ index of the first occurrence of the substring st in S, and -1 if ^{st is} not in S.

S.rfind(st) - Same as find, but starts from end.

- `s.index(st)` - Same as `find`, but raises an exception if `st` is not in `s`.
- `s.rindex(st)` - Same as `index`, but starts from the end of `s`.
- `s.lower()` - Converts all uppercase letters in `s` to lowercase.
- `s.replace(old, new)` - replaces all occurrences of the string `old` in `s` with the string `new`.
- `s.strip()` - removes trailing whitespace from `s`.
- `s.split(d)` - splits `s` using `d` as a delimiter. Returns a list of substrings of `s`.
eg. `'hi hey hello'.split(' ')`
o/p: `['hi', 'hey', 'hello']`

* DICTIONARIES :-

- * Objects of type `dict` are like lists except that we index them using keys.
- * Think of a dictionary as a set of key/value pairs. Literals of type `dict` are enclosed in curly braces, and each element is written as a key followed by colon followed by a value.

o/p: `Monthnum = {'Jan': 1, 'Feb': 2, 'Mar': 3}`
`print('2nd month is' + Monthnum[2])`
`diff = Monthnum['Mar'] - Monthnum['Jan']`
`print('diff is', diff)`
 o/p: 2nd month is Feb
 diff is 2

✓ The entries in a `dict` are unordered and can't be accessed with an index.

d = { 'Jan': 1, 'Jan': 2, 'March': 3 }

print(d['Jan']) => 2

d = { 'Jan': 1, 'Feb': 2 }

print(d[3]) => error

(ValueError: '3' is not a valid index)

So, Monthnum[1] refers to Jan rather than Feb.

Like list dictionary is Mutable.
We can add entry by writing,

Monthnum['June'] = 6

Or Change an entry by writing
Monthnum['March'] = 'V'

→ Dictionary reduce the difficulty of writing a variety of programs.

e.g. use dictionary to write a program to translate between languages.

If one line is so long and we want to continue next text of line then we use backslash \, to indicate continuation of previous line.

→ Side effect is Mutability.

e.g. Je boise du Vin rouge.

means → I good of wine red

now, FTOE ['bois'] = 'wood'

print(translate('Je boise du Vin rouge',
dicts, 'French to English'))

O/P:

I wood of wine red

→ There is no inbuilt functionality which provide Mapping from key to Values.

→ It is easy to implement a dictionary by using a list in which each element is a Key/value pairs.

We can write a function that does the associative retrieval.

eg., `def keysearch (L, k):`
 for elem in L:
 if elem[0] == k:
 return elem[1]

return none
 + but this $\propto^n$ is inefficient that in worst case it will have to check each element in a list to perform a single retrieval.

→ Built-in implementation is fast. It uses a technique called Hashing. which can do lookup in independent size of the dictionary.

* Creating Dictionary :

```
Dict = {}  
print ("empty dictionary")  
print (Dict)
```

Creating dictionary with integer keys

```
Dict = {1: 'ab', 2: 'x', 3: 'y'}  
print ("In dictionary with the use of integer keys")  
print (Dict)
```

Creating dictionary with Mixed keys

```
Dict = {'Name': 'xy', 1: [1, 2, 3, 4]}  
print ("In dictionary with mixed keys")  
print (Dict)
```

Creating dictionary with dict() method

```
Dict = dict({1: 'Geeks', 2: 'x', 3: 'y'})  
print ("In dictionary with the use of dict()")  
print (Dict)
```


Creating dictionary with each item as a pair
 Dict = dict([(1, 'xy'), (2, 'yz')])
 print("dictionary with each item as a pair:")
 print(Dict)

O/P:

Empty dictionary :

{ }

Dictionary with the use of Integer keys:

{ 1: 'ab', 2: 'x', 3: 'y' }

Dictionary with the use of Mixed keys:

{ 1: [1, 2, 3, 4], 'Name': 'xy' }

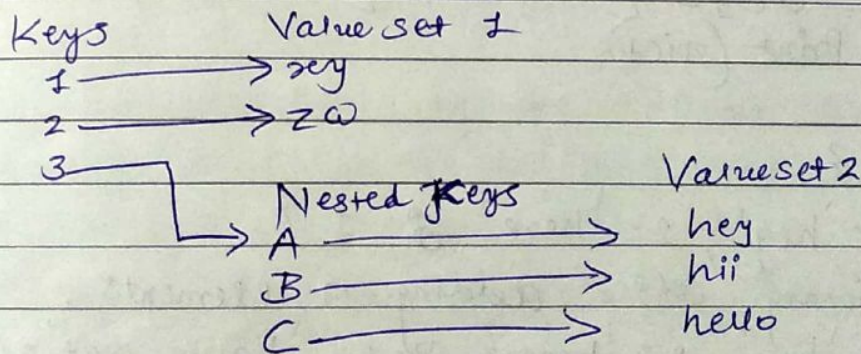
Dictionary with the use of dict()

{ 1: 'Geeks', 2: 'x', 3: 'y' }

Dictionary with each item as a pair:

{ 1: 'xy', 2: 'yz' }

* Nested Dictionary :



Dict = { 1: 'xy', 2: 'zw', 3: { 'A': 'hey', 'B': 'hi', 'C': 'hello' } }
 print(Dict)

O/P:

{ 1: 'xy', 2: 'zw', 3: { 'A': 'hey', 'B': 'hi', 'C': 'hello' } }

- updating an existing value in dictionary can be done by using the built-in update() method
- while adding a value, if the key value already exists, the value gets updated otherwise a new key with the value is added to the dictionary.

e.g.

Dict = {}

print(Dict)

Adding elements one at a time

Dict[0] = 'hey'

Dict[2] = 'here'

Dict[3] = 1

print(Dict)

Adding set of values to key

Dict['value-set'] = 2, 3, 4

print('in dictionary after adding 3 elements')

print(Dict)

updating existing key's value

Dict[2] = 'there'

print(Dict)

Adding Nested key value to the dictionary

Dict[5] = {'Nested': {'1': 'x', '2': 'y'}}

print(Dict)

O/P:

{}

{0: 'hey', 2: 'here', 3: 1}

dictionary after adding 3 elements:

{0: 'hey', 2: 'here', 3: 1, 'value-set': (2, 3, 4)}

{0: 'hey', 2: 'there', 3: 1, 'value-set': (2, 3, 4)}

{0: 'hey', 2: 'there', 3: 1, 5: {'Nested': {'1': 'x', '2': 'y'}}, 'value-set': (2, 3, 4)}

* Access the elements in Dictionary: →
 In order to access the items of a dictionary refer to its key name. Key can be used inside square brackets. There is also a

method `get()` can be used to access the elements.

e.g. `Dict = {'key': 'xyz', 'name': 'xyz', 3: 'hi'}`
 # accessing elements
`print('access element using key')`
`print(Dict['name'])`
`print(Dict[1])`
 # accessing element using `get()` method
`print(Dict.get(3))`

O/P: access element using key
 xyz
 xyz
 hi

* Deleting elements :

- deletion of key can be done by using the `del` keyword.
- using `del` keyword specific values or whole dictionary can be deleted.
- `pop()` and `popitem()` can also be used for deleting specific & arbitrary values from the dictionary.
- All the items from the dictionary can be deleted at ONCE using `clear()` method.
- `del Dict` will delete the entire dictionary.

e.g.

```
Dict = {5: 'welcome', 6: 'To', 7: 'xy',
        'A': {1: 'xy', 2: 'for', 3: 'xy'},
        'B': {1: 'xy', 2: 'wz'}}
print(Dict)
```


deleting a key value
del Dict[6]
print(Dict)

deleting a key from Nested dictionary
del Dict['A'][2]
print('deleting from Nested')
print(Dict)

deleting a key using pop()
Dict.pop(5)
print('deleting using pop')
print(Dict)

Dict.popitem()
print('In pops first element')
print(Dict)

deleting entire dictionary

Dict.clear()
print("entire dict is deleted")
print(Dict)

O/P:

{ 'A': {1: 'xy', 2: 'for', 3: 'xy'}, 'B': {1: 'xy', 2: 'wz'},
5: 'welcome', 6: 'To', 7: 'xy' }

After deleting 6 { 'A': {1: 'xy', 2: 'for', 3: 'xy'}, 'B': {1: 'xy', 2: 'wz'},
5: 'welcome', 7: 'xy' }

deleting from Nested { 'A': {1: 'xy', 3: 'xy'}, 'B': {1: 'xy', 2: 'wz'},
5: 'welcome', 7: 'xy' }

Popping 5 { 'A': {1: 'xy', 3: 'xy'}, 'B': {1: 'xy', 2: 'wz'}, 7: 'xy' }

Pops first element { 'B': {1: 'xy', 2: 'wz'}, 7: 'xy' }

entire dict is deleted
{ }

* Common Operations on Dicts :

`len(d)` - Returns the number of items in `d`.

`d.keys()` - Returns a view of keys in `d`.

`d.values()` - Returns a view of the values in `d`.

`k in d` - Returns True if key `k` is in `d`.

`d[k]` - Returns item `d` with key `k`.

`d.get(k, v)` - Returns `d[k]` if `k` is in `d`, and `v` otherwise.

`d[k] = v` - Associates the value `v` with key `k`. If there is already a value associated with `k`, that value is replaced.

`del d[k]` - Removes the key `k` from `d`.

`for k in d` - Iterates over the keys in `d`.

eg.

→ `d = {'Jan': 1, 'Feb': 'J'}`
`print(d['Jan'] - d['Feb'])` (`1 - J` # not possible)
O/P ⇒ error (TypeError)

→ `d = {'Jan': 1, 'Feb': 2}`
`print(d['Jan'] - d['Feb'])`
O/P = `-1` //

→ `d = {10: 1, 20: 2}`
`print(d[20] - d[10])`
O/P ⇒ `1`

→ `d = {10: 1, 2: 2}`
`print(d[2] - d[1])`
O/P ⇒ error

Ch. 6. Testing & Debugging

- * Testing $\Rightarrow$ Process of running a program to try and ascertain whether or not it works as intended.
- * Debugging $\Rightarrow$ process of trying to fix a program that you already know does not work as intended.
- * Testing $\Rightarrow$ purpose: to show that bugs exist
not to show that bug free.
- $\rightarrow$ The key to testing is finding a collection of inputs, called Test Suite, that has a high likelihood of revealing bugs.
- $\rightarrow$ A partition of a set divides that set into a collection of subsets such that each element of the original set belongs to exactly one of the subsets.
e.g. $\text{Isbigger}(x, y)$. The set of possible inputs is all pairwise combinations of integers. One way to partition this set is into these seven subsets:

x positive, y positive	x negative, y negative
x positive, y negative	x negative, y positive
$x=0, y=0$	$x=0, y \neq 0, x \neq 0, y=0$
- $\rightarrow$ If one tested the implementation on at least one value from each of these subsets, there could be reasonable probability of exposing a bug if one exists.
- $\rightarrow$ For most programs, finding a good partitioning of the input is far easier said than done. Typically people rely on heuristic based on exploring different paths through some

combination of code and specifications. Heuristic based on exploring paths through the code falls into a class called glass-box Testing.

→ Heuristic based on exploring paths through the specification falls into a class called Blackbox Testing.

* Black-Box Testing →

Blackbox Tests are Constructed without looking at the code to be tested.

→ It allows testers and implementers to be drawn from separate populations.

e.g.

When those of us who teach programming courses generate test cases for the problem sets we assign students, we are developing Black ~~box~~ Test Suites. Developers of commercial software often have quality assurance groups that are largely independent of development groups.

→ it reduces the likelihood of generating test suites that exhibit mistakes that are correlated with mistakes in the code.

e.g. Author of a program made the assumption that a f^n would never be called with a negative number. If the same person constructed the test suite for the program, he would likely repeat the mistake and not test the function with a negative argument.

→ Positive feature of blackbox testing is that it is robust with respect to implementation changes. As test data is generated without knowledge of the implementation.

→ The tests need not to be Changed when the Implementation is Changed.

→ a good way to generate black-box test data is to explore paths through a specification.
Consider specification,

```
def Sqrt(x, epsilon)
    """ Assumes x, epsilon float
        x >= 0, epsilon > 0 returns result such
        that x-epsilon <= result <= x+epsilon """
```

→ So, only two paths are Seemed through this Specification: one Corresponding to $x=0$ and Other $x>0$.

So, only these two test Cases are to be tested, it is hardly Sufficient.

→ Boundary Conditions Should also be tested.
When looking at lists this often means looking at empty list, a list with exactly one element, and a list containing lists.
when dealing with numbers Start with Smallest to the largest one.

e.g.

x	epsilon
0.0	0.0001
25.0	0.0001
0.5	0.0001
2.0	0.0001
2.0	1.0/2.0 ** 64.0
1.0/2.0 ** 64	1.0/2.0 ** 64.0
2.0 ** 64.0	1.0/2.0 ** 64.0
1.0/2.0 ** 64.0	2.0 ** 64.0
2.0 ** 64.0	2.0 ** 64.0

Fig. Testing boundary Conditions

- In fig. first rows represent typical cases. Value for x includes a perfect square, a number less than one and a number with irrational square root. If any of these test case fails, there is a bug in program, need to be fixed.
- Perhaps there is a bug in a code or the specification needs to be changed.
- It might be unreasonable to expect to find an approximation of square root when epsilon is too small.
- Another boundary condition to think about is aliasing.

def copy(L1, L2):

 """ Assumes L1 and L2 are lists

 Mutates L2 to be a copy of L1 """

 while len(L2) > 0: # remove elements from L2

 L2.pop() # remove last element of L2

 for e in L1: # append L1 to empty L2

 L2.append(e)

It will work when both lists are not same

* Glass-Box Testing :

- Black box Testing should never be skipped, but it is rarely sufficient.
- Without looking at internal structure of the code it is impossible to know which test cases are likely to provide new information.

e.g.

def isprime(x):

 """ Assume x is a positive int returns true if prime """

 if x <= 2:

 return False

CP Date _____
Page No. _____

```

for i in range(2, x):
    if x % i == 0:
        return False
return True

```

+ From code, the values 0, 1, 2 are treated as special cases, and therefore need to be tested. Without looking at code, one might not test `isPrime(2)` and would not discover that it returns false, erroneously indicating that 2 is not prime.

+ Glass box Test Cases are easier to Construct than black box suite because Specification (black box) are usually incomplete and often sloppy, making Challenge to estimate the inputs.

+ Where as in glass-box a notion of path is well defined and relatively easy to evaluate how one is exploring the space.

+ a glass-box test ^{suite} is Path Complete if it exercise every Potential path through the program, but it is impossible to achieve due to number of times each loop is executed and depth of each recursion.

e.g. factorial follows different path in recursion.

+ path Completion does not guarantee that all bugs will be exposed.

e.g.

```

def abs(x):
    """
    Assume x is an int returns x if x >= 0
    And -x otherwise """
    if x < -1:
        return -x

```


else :

return x

- The Specification Suggests that there are two possible Cases: x either is negative or not. So, it suggests only $\{2, -2\}$ is sufficient to explore all the paths in the specification.
- This two cases go through all the paths. So, it is a Complete glass box testing suite.
- but the problem with this is that it will not expose the fact that $\text{abs}(-1)$ will return -1 .
- despite of the limitations, there are few rules of thumbs that are worth to follow:
 - exercise both branches of all if stmts.
 - Make Sure that each except clause is executed.
 - for each for loop, have test cases in which,
 - The loop is not entered,
 - The body of the loop is executed at least once.
 - The body of the loop is executed ~~at least~~ ^{more than} once.
 - for each While loop,
 - Same cases as have for for loop.
 - include test cases corresponding to all possible ways of exiting the loop.
- e.g. while $\text{len}(L) > 0$ and not $L[i] == e$
find cases where the loop exits because $\text{len} > 0$ and $L[i] \neq e$.
- for recursive functions, include test cases that cause the function to return with no recursive calls, exactly one call, more than one call.

* Conducting Tests :→

Testing Occurs in Two Phase: Unit Testing Integration Testing

- Start with unit Testing. During it Tester constructs and run tests designed to ascertain whether individual units of code (f^n) work properly.
- It is followed by Integration Testing, whether the program as whole behaves as intended or not.
- * but problem in Integration Testing, leads to make changes in individual units.
- Integration Testing is more Challenging than Unit Testing.
- Problems of scale can make integration Testing difficult.
- Many Industrial software development organizations have a software quality assurance group that is separate from the group charged with implementing the software. The mission of this group is to ensure that before the release of software it is suitable for its intended purpose.
- Sometimes development group is responsible for unit testing and QA group for integration testing.
- In Industry, the testing process is automated. Tester don't do for individual input, They use Test Drivers that autonomously,
 - Setup environment needed to invoke the unit / program to be tested.
 - Invoke the program with predefined or automatically generated ^{sequence of} inputs.
 - Save the results of invocations.

- Check the acceptability of the result of the tests
- And prepare appropriate report.
- during unit testing, we need to build Stub as well as Drivers
- Drivers: → It simulate parts of the program that use the unit being tested
- STUBS: → It simulate parts of the program used by the unit being tested.
- Stubs are useful because they allow people to test units that depend upon software or sometimes even hardware that does not yet exist.
- e.g. Stubs are used in Topdown approach, when you have the major module ready to test but submodules are not ready yet. So, simple language stubs are "called" programs, which are called in to test the major module's functionality.
- We have 3 modules; login, home, user. now only login module is ready for test, so we write dummy code for home & user which are not ready to run the login page.
- Drivers are calling programs used in bottomup testing approach, which is used when submodules are ready but main module is not ready.
- So, home and user is ready but login is not ready. So, make dummy module for login, which is called as Driver.
- one attraction of automating the testing process is that it facilitates Regression Testing.

Whenever any change is made, you should check that the program still passes all of the tests that it used to pass.

* Debugging :-

→ The process of fixing flaws in software came to be known as Debugging.

→ If your program has a bug that means you made a mistake. Runtime bugs can be categorized along two dimensions:

* Overt → Covert :

best bugs { Overt bug has an obvious manifestation. e.g. The program crashes or takes for longer (may be forever) to run than it should.

→ Covert bug has no obvious manifestation. e.g. Program may run to conclusion without problem.

→ Many bugs fall between the two extremes.

* Persistent → Intermittent :-

→ Persistent bug occurs ~~when~~ everytime the program is run with the same inputs.

→ Intermittent bug occurs only when same input with same conditions

→ Good programmers try to write their programs in such a way that programming mistakes lead to bugs that are both Overt & Persistent. This is often called defensive programming.

→ The best kinds of bugs to have are Overt & Persistent. Developers can be under no illusion

about the advisability of deploying the program.
 + Perhaps the program will do something horrible before crashing. e.g., delete files, but at least the user will have reason to be worried (if not panicked).

* Debugging Tools :->

There are many debugging Tools, some of which are ~~built~~ built into IDE like pycharm and other standalone applications.

- + Pdb is a debugger built into the Python standard library.
- + Web-pdb provides a web based user interface for pdb to make it easier to understand what is happening while running the code.
- + Wdb uses Websockets to allow you to debug running Python code from a web browser.
- + Pyflame (source code) is a profiling Tool that generates flame graphs for executing Python program code.
- + Objgraph (source code) uses graphviz to draw the connection between Python objects running in an application.

e.g.

```
import pdb
a = "aaa"
Pdb.set_trace()
b = "bbb"
c = "ccc"
final = a + b + c
Print final
```

O/P:

```
-> Pdb.set_trace()
(Pdb) p a
'aaa'
(Pdb) p final
'aaa bbb ccc'
```


Exceptions & Assertions

CP

Date _____

Page No. _____

→ An "exception" is usually defined as "something that does not conform to the norm", and is therefore somewhat rare. There is nothing rare about exceptions in Python.

e.g.

```
Test = [1, 2, 3]
Test[3]
```

The interpreter will respond with something like
`IndexError: list index out of range`

→ Most common types of exceptions are `TypeError`, `IndexError`, `NameError`, `ValueError`.

→ Handling Exceptions :

When an exception is raised that causes the program to terminate, we say that unhandled exception has been raised.

✓ exceptions when raised, can and should be handled by the program.

e.g. Programmer tries to divide number by 0
he tries to open a file which does not exist
So,

```
try:
    ratio = numSuccess / numFail
except ZeroDivisionError:
    print("numFail = 0")
```

→ In this `numFail = 0` then exception raises and control immediately jumps to the `except` block.

* Polymorphic function → the fⁿ who works for arguments of different types.

→ We can write separate `except` block for each kind of exception, which allows the program

to choose an action based upon which exception raised.

e.g. except:

The except block will be entered if any kind of exception is raised within the try block.

* Exception as Control Flow Mechanism : *

* don't think of exceptions as purely for errors. They are convenient flow-of-control mechanism that can be used to simplify programs.

→ In python, it is more usual to have a function raise an exception when it can't produce a result that is consistent with the function's specification.

✓ → The python raise statement forces a specified exception to occur. The form of a raise statement is:

raise exceptionname (arguments)

↓
inbuilt exceptionname

e.g. ValueError

↓
Mostly single String

→ Programmers can define new exceptions by creating a subclass of built-in class exception.

ex. Implement a function that satisfies the Specification : *

```
def findEven(L)
```

"""

Assumes L is a list of integers
Returns the first even num in L

Raises ValueError if L does not
contain an even number """

using exception for Control-flow

→ Consider the function definition:

```
def getratios (vect1, vect2):
```

""" Assumes: vect1 and vect2 are equal length lists of numbers

Returns: a list containing the ^{meaningful} values of $\text{vect1}[i] / \text{vect2}[i]$ """

```
    ratios = []
```

```
    for index in range(len(vect1)):
```

```
        try:
```

```
            ratios.append(vect1[index] / vect2[index])
```

```
        except ZeroDivisionError:
```

```
            ratios.append(float('nan')) # nan = not a number
```

```
        except:
```

```
            raise ValueError('getRatios called with bad arguments')
```

```
    return ratios
```

→ There are two except blocks associated with the try block. If an exception is raised within the try block, python first checks to see if it is a ZeroDivisionError. If, so, it appends a special value, nan (not a number) of type float to ratios. If the exception is anything other than ZeroDivisionError, the code ~~executes~~ the second except block which raises a ValueError exception with an associated string.

* Assertions :

The Python Assert Statement provides Programmers with a simple way to confirm that the state of a computation is as expected. assert statement can take one of two forms:

assert Boolean expression

or

assert Boolean expression, Argument

When an assert statement is encountered, the Boolean expression is evaluated.

If evaluated to True, execution proceeds on its merry way. If it evaluates to false, an ~~AssertionError~~ exception is raised.

+ Assertions are useful defensive programming tool. They can be used to confirm that the arguments to a function are of appropriate types. They are also a useful debugging tool. They can be used to confirm that intermediate values have the expected values or that a fn returns an acceptable value.

Eg. `def avg(marks):`
 `assert len(marks) != 0`
 `return sum(marks)/len(marks)`
 `marks = [ ]`
 `print ("average: ", avg(marks))`

Error:
 `assert len(marks) != 0`
 AssertionError.