Python Programming Language

Dr. Ami Tusharkant Choksi
Associate Professor,
Computer Engineering Department,
C.K.Pithawala College of Engineering & Technology, Surat, Gujarat State, India.
email: ami.choksi@ckpcet.ac.in

April 25-May 01, 2020

Contents I

Contents II

Python Certifications i hold I

1. Secured 82% i.e. Silver medal in NPTEL and IIT Madras course on “Python for Data Science”, 2019. Secured position in Top 6% of certified candidates.
2. Secured Gold Medal in NPTEL and IIT Ropar initiated course titled “Joy of Computing Using Python”, 2018. Secured Top 6% in the NPTEL certification exam of the course.
3. Secured 1st Rank in the Lecture and Practice programs Editing contest of NPTEL course “Joy of Computing using Python”, 2018 by IIT Ropar & NPTEL.
4. Cleared “Crash Course in Python (In Hindi)”, of E&ICT, IIT Kanpur course with 86%, 2018.
5. Cleared “Python Programming – A Practical Approach”, of E&ICT, IIT Kanpur course with 88.02%, 2018.

Introduction to Python I

- **Python is Interpreted** - Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive** - One can interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** - Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Easy-to-learn** - Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** - Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** - Python's source code is fairly easy-to-maintain.

Introduction to Python II

- A **broad standard library** - Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** - Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable** - Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** - You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases** - Python provides interfaces to all major commercial databases.

Introduction to Python III

- **GUI Programming** - Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable** - Python provides a better structure and support for large programs than shell scripting.
- **Easy integration** - It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.
- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.

Introduction to Python IV

- It supports automatic garbage collection.
- It supports exception handling.

Install Python I

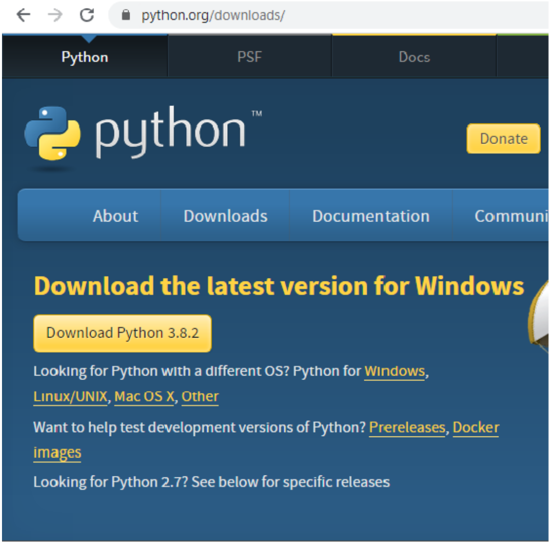
Listing 1: Windows

```
- From https://www.python.org/downloads/ download the exe file
of latest python version and install that.
```

Listing 2: Ubuntu

```
- From terminal,
sudo apt-get install python3.8
Or
- From https://www.python.org/downloads/ download the latest
version's tar.gz file.
+ Extract that and move the python directory to /usr/local
if you have superuser access
+ Set the path of /usr/local/python-version/bin to system's path
- Check whether python is installed or not using
rpm -qa python
```

Install Python II



Download the latest version for Windows

Download Python 3.8.2

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

Want to help test development versions of Python? Prereleases, Docker images

Looking for Python 2.7? See below for specific releases

Figure: Python Download Page

Typical way of Programming

- Write Program
- Run it with some input
- Is output OK?
- If No, Write Program
- If yes, more inputs? If no, *Finish*
- If yes, more inputs? If yes, Run it with some input
- If no, Write Program.

Start Programming with Python I

- Begin any text editor and create test.py, e.g.vi test.py with the following content
- Extension is always .py

Listing 3: First Program

```
#!/usr/bin/python
print("Hello World")
```

- Run the python file on terminal using

```
python test.py
```

Listing 4: Output of test.py

```
Hello World
```

Python Language Editors

- Sublime text : `https://www.sublimetext.com/3`
- Spyder : apt-get install spyder
- Python IDLE : sudo apt-get install idle
- PyCharm : `https://www.jetbrains.com/pycharm/download/#section=linux`
- Jupiter Notebook : https://www.anaconda.com/distribution/

Use of Print I

Listing 5: Print value of a variable

```
var=2
print("var",var)#output is 2
print("2 + 2 =", 2+2) #output is 4
print ("2*3=", 2*3, "4*5", 4*5)#output is 20
```

Listing 6: Output

```
var 2
2 + 2 = 4
2*3= 6 4*5 20
```

Input I

```python
num = input("Enter your age : ")
print("your age is ", num)

'''Output
Enter your age : 5
your age is 5
'''
```

- In Python, input function always reads data in string format

Elements of Python I

- A Python program is a sequence of definitions and commands
- Commands manipulate objects
- Each object is associated with a Type

Standard Data Types in Python Langauge I

- Numbers: int, float, long
- String: Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result \ can be used to escape quotes:
- List: Consists of a number of values separated by commas, in squared bracket. It is an ordered collection of values.
- Tuple: Consists of a number of values separated by commas
- Sets: They are unordered collection of data separated by comma. Union, intersection, difference like operations can be operated on sets.
- Dictionary: It is a set of key: value pairs, with the requirement that the keys are unique.

Major types in Python I

- Scalar and Non-scalar are the major types in Python
- Scalar: Objects that do not have internal structure
- int(signed integers), float(floating point), bool(Boolean), *NoneType* are scalar types
- NoneType is a special type with a value
- The value is called *None*
- Non-scalar: Objects having internal structure
- String, lists, tuples, dictionaries, sets and user defined classes are non-scalar type

Find a type of a number I

Listing 7: Find a type of a number

```
#Type of var/value
print(type(5)) #<class 'int'>
print(type(-10))#<class 'int'>
print(type(10.5))#<class 'float'>
print(type(True))#<class 'bool'>
print(type("hi ami"))#<class 'str'>
print(type(3!=2))#<class 'bool'>
```

Listing 8: Output

```
<class 'int'>
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'str'>
<class 'bool'>
```

Type Conversion I

- Convert type one to other.
- int of 3.6 is 3 or rounded to 4
- float value of 3 is 3.0

Listing 9: Type Conversion

```
print( int(2.5))#Output 2
print( int(2.3))#Output 2
print( int(3.9))#Output 3
print( float(3))#Output 3.0
print( int(’75’))#Output 75
print(int(’abcd’))
#Output
#Error:::Traceback (most recent call last):
#File "<pyshell#5>", line 1, in <module>

int(’abcd’)
```

Type Conversion II

```
#Output
#ValueError: invalid literal for int() with base 10: ’abcd’

a=input(’enter no : ’)
print("adding 5 to a ", a+5)

#Output:
#enter no : 7
#Traceback (most recent call last):
#File "<pyshell#10>", line 1, in <module>
#print("adding 5 to a ", a+5)
#TypeError: must be str, not int

#Modified one:
print("adding 5 to a ", int(a)+5)
#Output
#adding 5 to a 12

```

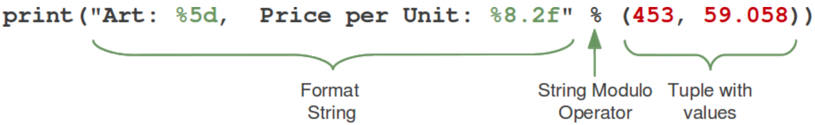
Type Conversion III

```
int(’7.59’)
#Will return error, as argument to int is not an valid integer.

a=10
b="ami"
c=str(a)+b
print(c)
#Output
#’10ami’
```

Formatted Print I

- Formatted print, as we used to have in *C language*
- %5d and %8.2f



```
print("Art: %5d,  Price per Unit: %8.2f" % (453,  59.058))
```

Figure: Formatted Print

- %5d - total space allocated is 5, as 5 is positive number, the number is printed to the right hand side, as shown below.

		4	5	3
1	2	3	4	5

- %8.2f - total space allocated is 8 from which, one is for decimal point, .2f, so after decimal point 2 places and other number will be printed in 5 places, number is printed from the right hand side

Formatted Print II

			5	9	.	0	6
1	2	3	4	5	6	7	8

- -%5d (negative 5) - - total space allocated is 5, as 5 is positive number, the number is printed to the left hand side, as shown below.

4	5	3		
1	2	3	4	5

- -%8.2f (negative 8.2) - - total space allocated is 8 from which, one is for decimal point, .2f, so after decimal point 2 places and other number will be printed in 5 places, number is printed from the left hand side

Conversion	**Meaning**
d	Signed integer decimal.
i	Signed integer decimal.
o	Unsigned octal.

Formatted Print III

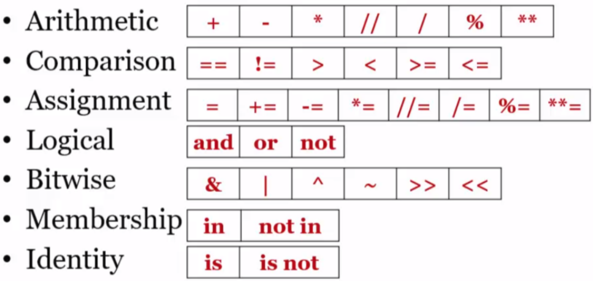
u	Obsolete and equivalent to ’d’, i.e. signed integer decimal.
x	Unsigned hexadecimal (lowercase).
X	Unsigned hexadecimal (uppercase).
e	Floating point exponential format (lowercase).
E	Floating point exponential format (uppercase).
f	Floating point decimal format.
F	Floating point decimal format.
g	Same as "e" if exponent is greater than -4 or less than precision, "f" otherwise.
G	Same as "E" if exponent is greater than -4 or less than precision, "F" otherwise.
c	Single character (accepts integer or single character string).

Formatted Print IV

r	String (converts any python object using repr()).
s	String (converts any python object using str()).
%	No argument is converted, results in a "%" character in the result.

Operators I

- We want to learn about different types of operators in Python



• Arithmetic	+	-	*	//	/	%	**	
• Comparison	==	!=	>	<	>=	<=		
• Assignment	=	+=	-=	*=	//=	/=	%=	**=
• Logical	and	or	not					
• Bitwise	&	|	^	~	>>	<<		
• Membership	in	not in						
• Identity	is	is not						

Figure: Operators in Python

- Operators are used with variables/identifiers
- Variable is named with digits, underscore(_) and numbers
- Variable name must start with letter or underscore(_)

Operators II

- Valid identifiers names are area, radius5, max_profit, fUn
- Invalid identifiers names are 5j, max profit, lab.7
- Certain names are reserved in Python, which has special meaning and cannot be used as an identifier
- Some reserved words : False, True, class, in, is, and, or, import, return
- The language is case sensitive
- i.e. Acad $\neq$ acad $\neq$ ACADS
- Choose meaningful name for identifier. e.g.for counter it should be count instead of temp1
- Not too long names or not too short names should be used for identifier. e.g. Max is preferable over Maximum
- Identifiers are unlimited in length. But style guide of Python i.e.PEP-8, limit all lines to a maximum of 79 characters.

Statement and Expression I

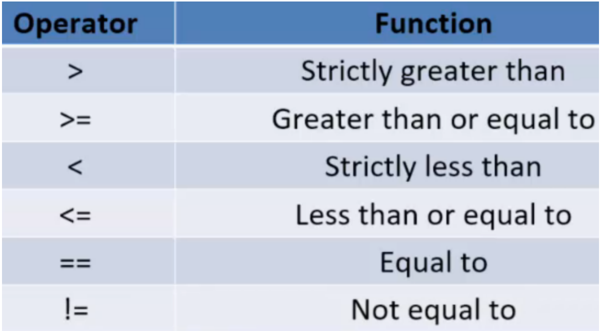
- A statement is a complete line of code that performs some action, while an expression is any section of the code that evaluates to a value.
- Every expression can be used as a statement (whose effect is to evaluate the expression and ignore the resulting value), but most statements cannot be used as expressions.
- Expression uses different types of operators in them.

Assignment I

- Assignment operator is used as, variable = value
- RHS of '=' is evaluated first and then assigned to LHS variable name
- e.g. a = 5
- a = a + 1, i.e. 5 + 1 is evaluated as 6, and the value is assigned to a.
- / - simple division, whatever is the result
- // - integer division
- e.g. a=4/5 = 0.8
- e.g. 4//5 = 0
- 9/3 = 3.0, so type(9/3) is float
- type('None') is a string
- When more than one operators are there in an expression, which operator will be evaluated first is to be decided. It is using *operator precedence*.

Boolean Expression I

- An expression that evaluates to either True or False.
- Relational operators compares two quanties.



Operator	Function
>	Strictly greater than
>=	Greater than or equal to
<	Strictly less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

Figure: Relational Operators

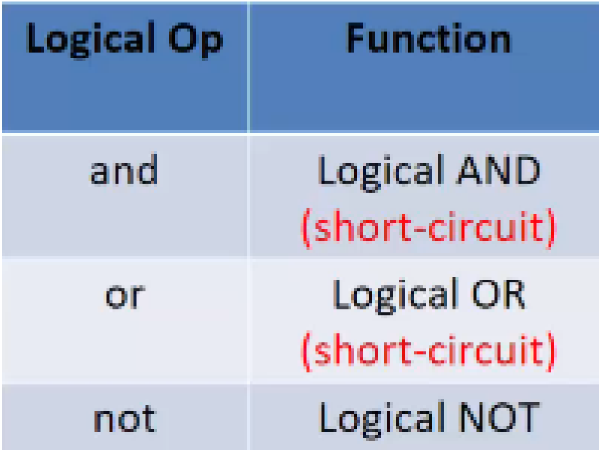
- e.g.3>2 True

Boolean Expression II

- ’z’>’a’ True
- ’car’ < ’carpet’ True
- 5<10 True

Logical Operators I

- Logical Operators are 'and' and 'or' and 'not'



Logical Op	Function
and	Logical AND (short-circuit)
or	Logical OR (short-circuit)
not	Logical NOT

Figure: Boolean Expression

Logical Operators II

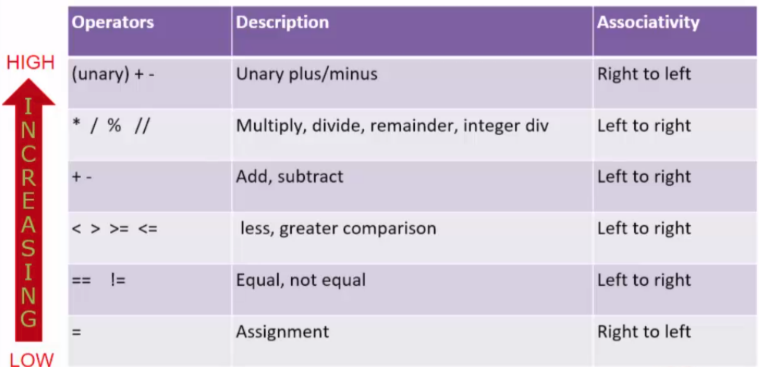
Complex logical expressions involves some combinations of arithmetic, logic, relational operators, constants and function calls.

e.g. (x + 7>93) and not(y+3<z) or (abs(sqrt(w)-g)<epsilon)

Precedence of Operators I

- Parantheses is having higher precedence.
- Unary operators + and - have next higher precedence.
- As we go from top->bottom in the table, operator precedence is increasing.
- Python evaluates expressions from left to right. *Notice* that while evaluating an assignment, the right-hand side is evaluated before the left-hand side.
- Other precedences are as in figure.

Precedence of Operators II


HIGH
INCREASING
LOW

Operators	Description	Associativity
(unary) + -	Unary plus/minus	Right to left
* / % //	Multiply, divide, remainder, integer div	Left to right
+ -	Add, subtract	Left to right
< > >= <=	less, greater comparison	Left to right
== !=	Equal, not equal	Left to right
=	Assignment	Right to left

Figure: Operator Precedence

Operator	Description
lambda	Lambda expression

Precedence of Operators III

if - else	Conditional expression
or	Boolean OR
and	Boolean AND
not x	Boolean NOT
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparisons, including membership tests and identity tests
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
+, -	Addition and subtraction

Precedence of Operators IV

*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, remainder
+x, -x, ∼x	Positive, negative, bitwise NOT
**	Exponentiation
await x	Await expression
x[index], x[index:index], x(arguments...), x.attribute	Subscription, slicing, call, attribute reference
(expressions...), [expressions...], {key:value...}, {expressions...}	Binding or tuple display, list display, dictionary display, set display

Precedence of Operators V

Listing 10: Precedence of Operators

```
e.g. x = -5*4//2*3+-1*2
= -20*3-2
=-30-2
=-32
```

Control Statements and Loops

- Conditionals and loops we want to learn

Conditionals I

- Python uses if-else for conditionals.
- e.g. Compare two numbers x & y, print min

Listing 11: if-else example

```
x,y = 5,10
if x<y:
       print(x)
else:
       print(y)
print(’is the minimum’)

’’’
Output is
5
is the minimum’’’
```

- No explicit bracket

Conditionals II

- Identation is important in Python

Listing 12: if-else example

```
x,y = 5,1
if x<y:
              print(x)
else:
              print(y)
print('y')
print(' is the minimum')
'''
Output is
1
y
is the minimum  '''
```

Conditionals III

Listing 13: bested if-else

```
x,y,z = 115,100,150
if x<=y:
              if x<=z:
                     print(x)
              else:
                     print(z)
else:
              if y<=z:
                     print(y)
              else:
                     print(z)
'''
Output
100'''
```

- This was nested if-else

Conditionals IV

Listing 14: if-elif-else

```
x,y,z = 115,100,15
if x<=y and x<=z:
       print(x)
elif y<=x and y<=z:
       print(y)
else:
       print(z)
'''
Output
15'''
```

Listing 15: Precedence example

```
print(3*20/5/4-17-30+45+7+2)
#Output: 10.0
```

Conditional Example:Float Representation I

Listing 16: Conditional Example:Float Representation

```
x = 0.1*3
print(x)
if x==0.3:
        print("Fly")
else:
        print("Sleep")

#Output
#0.30000000000000004
#Sleep
```

- Logically, we think, the above program should output, Fly, but internal representation of float is not 0.3
- We printed x its value is not 0.3 exact
- Floating points are representeed as approximations of numbers.

Conditional Example:Float Representation II

- Instead of x==y for floating point comparison, abs(x-y <= epsilon) can be used, where *epsilon* is suitably chosen small value
- Because of the approximations, comparison of floats is not exact

Listing 17: or condition

```
if (5 > 6) or (6 > 5):
       "ok"
else:
       "not ok"
#Output:
nothing is printed
```

Conditional Example:Float Representation III

Listing 18: not condition

```python
y = 0.1*3
if y != 0.3:
       print ('Launch a Missile')
else: print ("Let's have peace")
#Output:
#Launch a Missile
```

Loops I

- for loop has range in it.
- range(s,e,d): s-starting, e-end, d-distance
- range(s,e) : s-starting, e-end, d-1
- range(e): s-0, e-end, d-1

Listing 19: for loop syntax

```
for iterating_var in sequence:
        statements(s)
```

Loops II

Listing 20: for loop with range

```
n=5
for i in range(1, 11):
        print(n,"*",i,"=",n*i)

#Output
'''
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

Loops III

```
'''
```

Listing 21: for example

```
for i in range(1,5):
        print(1.0/i)

'''Output:
1.0
0.5
0.3333333333333333
0.25
'''
```

Listing 22: While syntax

```
while expression:
        statement(s)
```

Loops IV

Listing 23: while loop

```python
count=0
while count<10:
       print(count)
       count=count+1
#Output: 1.2.3.4.5.6.7.8.9.10.
```

Listing 24: While else loop syntax

```
while expression ":" suite
["else" ":" suite]
```

Loops V

Listing 25: While else Example

```python
#while-else
a = eval(input("Enter number : "))
while a>10:
        print(a,end=".")
        a=a-1
else:
        print("\nFalse condition or loop")
#Output:
#Enter number : 15
#15.14.13.12.11.
#False condition or loop
```

Loops VI

Listing 26: for example with Tuple

```
for i in (1,2,3,4,5):
        print(i)
'''Output:
1
2
3
4
5
'''
```

Loops VII

Listing 27: "Geometric Progression"

```
'''Given positive real numbers r and a, and a positive integer, n,
r = float(input('r= '))
a = float(input('a= '))
n = int(input('n = '))

term = a
print(term)
for i in range(1,n):
        term = term * r #(i+1)th term
        print(term)#display (i+1)th term

'''Output
r= 5
a= 6
n = 7
6.0
```

Loops VIII

```
30.0
150.0
750.0
3750.0
18750.0
93750.0
'''
```

Nested loop I

- Loop within a loop

Listing 28: Nested loop

```
for i in range(1,9):
        for j in range(1,6):
                print (i*j, end=’ ’)
        print()
'''Outut
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
5 10 15 20 25
6 12 18 24 30
7 14 21 28 35
8 16 24 32 40
'''
```

Nested loop II

Listing 29: Nested loop with range

```
for i in range(1,4):
       for j in range(i, 2*i):
              print(j, end='.')

'''Output:
1.2.3.3.4.5.
'''
```

Break and continue I

- Break is used for exiting the current while or for loop forceibly
- Continue is used for skipping an iteration of a loop, loopis not exited.

Listing 30: while loop

```
i=0
while i<=10:
       i=i+1
       if i%2==0:
              continue
       else:
              print(i, end=’ ’)
’’’Output:
1 3 5 7 9 11
’’’
```

Break and continue II

Listing 31: break

```
r,a,n = 2,10,20
term = a
for i in range(n):
       term = term * r
       if (i > 3):
              break
print (term)
#Output
#320
```

Functions I

- An independent, self-contained entity of a program that performs a well-defined task.
- Its has
  - Name: Function name
  - Arguments: to pass information to function
  - Body: logic of the function
  - Return Value: after calculation, the value that is given to the caller of the function

Listing 32: "add function"

```
#defining method/function
def add(a,b):
       return(a+b)
#calling method/function
print(add(5,4))
#Output
9
```

Functions II

- Function allows us to break complex problem in small sub-problems.
- Function allows to hide the details: i.e. Abstraction

Listing 33: printString function

```python
def printString(str):
       print(str)

printString("Hello World")
#Output
#Hello World
```

Functions III

Listing 34: find max of 4 numbers-Nice Question

```
x1=5
x2=3
x3=10
x4=1
y1 = min(x1, x2) #min
y2 = min(x3,x4) #min
y3 = x1 + x2 - y1 #min is removed, max(x1, x2) remains
y4 = x3 + x4 -y2#min is removed, max(x3, x4) remains
y5 = max (y3, y4) #max(max(x1,x2),max(x3,x4())
print("max of x1,x2,x3,x4: ",y5)
#Output
#max of x1,x2,x3,x4: 10
```

- Actual and formal parameters
- Execution of function using stack.

assert keyword I

- The `assert` keyword is used when debugging code.
- The assert keyword lets you test if a condition in your code returns True, if not, the program will raise an `AssertionError`.
- if condition returns **True**, then nothing happens
- if condition returns False, `AssertionError` is raised

Listing 35: assert example

```
x = "hello"

#if condition returns True, then nothing happens:
assert x == "hello"

#if condition returns False, AssertionError is raised:
assert x == "goodbye"#AssertionError is raised
```

assert keyword II

Listing 36: if condition is false, AssertionError is raised plus we can provide message

```
x = "hello"

#if condition returns False, AssertionError is raised:
assert x == "goodbye", "x should be ’hello’"
#Output:
#AssertionError: x should be ’hello’
```

Scope Rules I

- Scope of a name is the part of the program in which the name can be used. Here name can be name of identifier/function...etc.
- Two variables can have the same name ONLY if they are declared in separate scopes.
- A variable cannot be used outside its scope.
- Let's learn static scoping or lexical scoping
- Scope Rules of Functions:
- The scope of variables available in the argument list of a function's definition is the body of that function.
- If a variable is assigned within a function, its scope is the body of the function.
- A name is added to the scope associated with a function ONLY under the above two cases.

Scope Rules II

Listing 37: define min and max functions

```
def max(a1,b1):
        m1=0
        if(a1>b1):
                m1=a1;
        else:
                m1=b1
        return m1

def min(a2,b2):
        m2=0
        if(a2<b2):
                m2=a2;
        else:
                m2=b2
        return m2

```

Scope Rules III

```
#call functions
print(max(5,7))#7
print(min(5,7))#5
```

- scope of *m1, a1, b1* is within the body of a *function max*
- scope of *m2, a2, b2* is within the body of a *function min*

Scope Search I

- Python provides the support of function inside other function, i.e.Nested function.
- Suppose x is used in body of function *f*
- *x* is not found in body of function *f*
- It will be searched in function *g*, in which the function *f* was defined.
- Still if it is not found, it will be searched in the functions which encloses the function, that has used *x*, If x is not found, error is produced.

Scope Search II

Listing 38: Nested Function Example

```python
def outerFunction(text):
       text = text

       def nestedFunction():
              print(text)

       nestedFunction()

outerFunction("Hi")
```

Listing 39: Output of Nested Function Example

```
Hi
```

Call by value I

- Int, float and bool - call by value
- String, list, tuple - collection types- they are call by reference-i.e.object as reference

Listing 40: Call by value

```python
#call by value
def myfun(name, role):
        print(name,":",role)
#fucntion call
myfun(name="ami", role="mentor")

def main():
        myfun(name="aayushi",role="teacher")
        myfun("Pruthvi","Raja")
#fucntion call
        myfun(name="abhay", role="business man")
main()
```

Call by value II

```
#o/P:
#ami : mentor
#abhay : business man
#aayushi : teacher
#Pruthvi : Raja
```

Listing 41: Call by reference

```
#Call by reference
#Call by reference
def student(*arg):
       for i in arg:
              print(i,end=":")
       print()
def main():
#this will not work
#student(fname="ami",lname="choksi")
#this will work
```

Call by value III

```
        student(10)
        student(10,20)
        student(10,20,30)
main()
#O/P:
#10:
#10:20:
#10:20:30:
```

Globals I

- Globals allow functions to communicate with each other indirectly without parameter passing/return value
- Convenient when two functions want to share the data - without any direct caller/callee function
- If a function has to update a global, it must redeclare the global variable with global keyword.

Listing 42: Global Variable Example

```
COUNT = 5

def increment():
        global COUNT
        COUNT = COUNT+1

def decrement():
        global COUNT
```

Globals II

```
        COUNT = COUNT-1

increment()
print(COUNT)
decrement()
print(COUNT)
#Output
6
5
```

Listing 43: Scope Example

```
FACTOR = 10
def f(r):
        return FACTOR * r
def update_FACTOR():
        FACTOR = 16

```

Globals III

```
x =f(5)
update_FACTOR()
y = f(5)
print (x, y)

#Output
50 50
```

Listing 44: Passed Parameter Scope

```
M = 10

def double(M):
        return M+M

print (double(7), double(M))
#Output
14 20
```

Data Types in Details

- Number data type we have seen.
- Now we want to learn String, Tuple, List, Set and Dictionary.

Strings I

- String have type *str* in Python
- String are enclosed in single quote(') or double quote(")
- Backslash (
) is used to escape the quotes and special characters.
- len(String) is used to find length of the string
- + is used to concatenate two string.
- * is used to repeat a string, an *int* number of times.

Listing 45: String Operations

```python
#String Operations
name="ami choksi"

#length of the string
print(len(name))#Output 10
print(len('1\n2'))#Output 3
```

Strings II

```

#Concatenate and Repeat
designation = name + "’s designation is associate professor"
print(designation)#ami choksi’s designation is associate professor

#n times Repeatation - *n
print(name*5)#ami choksiami choksiami choksiami choksiami choksi
```

Listing 46: split()

```
#split()
b="my:name:is:ami"
c=b.split(’:’)
print(c)#[’my’, ’name’, ’is’, ’ami’]
```

Strings III

Listing 47: upper() lower()

```
#upper() lower()
s="ami"
print(s)#’ami’
up=s.upper()
print(up)#’AMI’
ls=s.lower()
print(ls)#’ami’
```

Listing 48: "Special Cases"

```
print(len(’Amy’s’ + ’Pen’)) #Error as single quoted string needs to
print(len("Amy\’s" + "Pen"))#Output: 8, as single quote inside stri
print(len("Amy’s" + "Pen"))#Output: 8, as double quoted string by d
```

String and Indexing I

- Strings can be indexed
- Negative indices start counting from the right
- Negative index start from -1
- if very big or very small number is given i.e. not in the range of String length, it gives error string index out of range

Listing 49: String indexing

```
name='Acads'
print(name[0])

print(name[3])
print(name[-1])
print('Hello'[1])
'''Output
A
d
```

String and Indexing II

```
s
e
'''

#print(name[50])#string index out of range
```

String Slicing I

- To obtain a substring
- s[start:end] starts at start and ends at end-1
- s[0:len(s)] is same as s
- Both start and end are optional
- s[:] is same as s[0:len(s)]

Listing 50: String Indexing

```
name=’amichoksi’
#print all characters from 0 to 3-1
print(name[0:3])#ami
#print all characters before 3rd index
print(name[:3])#ami
#print all characters after and including 3rd index
print(name[3:])#choksi
#print all characters
print(name[:])#amichoksi
```

String Slicing II

- When start and end have same sign, if start>=end, empty slice is returned, as shown in above example
- `print(name[-1:-4])`
- We want to look for Out of range slicing

a	m	i	c	h	o	k	s	i
0	1	2	3	4	5	6	7	8
-9	-8	-7	-6	-5	-4	-3	-2	-1

Listing 51: More Slicing

```python
name='Amichoksi'
# 012345678
#-987654321
print("-4:-1",name[-4:-1])# oks

print("-4",name[-4:])#oksi

```

String Slicing III

```
print("-4:8",name[-4:8])#oks

#if you give name[-4,4] it will not print anything as indexes s
print("-1:-4",name[-1:-4])#we have to go in positive direction,

print("1:-1",name[1:-1])#michoks
print("-10,-1",name[-10:-1],":" )#Amichoks

print("-50,20",name[-50:20])#Amichoksi
#When left index is very very less than the start index and rig

'''Output:
-4:-1 oks
-4 oksi
-4:8 oks
-1:-4
1:-1 michoks
-10,-1 Amichoks :
```

String Slicing IV

```
-50,20 Amichoksi
'''
```

Listing 52: Pass String to funcion

```
#pass String to funcion
#print all words of a string
def printWords(s):
       words=s.split()
       for i in words:
              print(i)

st="Hello and welcome to the world of computers"
printWords(st)

#Output:
#Hello
#and
```

String Slicing V

```
#welcome
#to
#the
#world
#of
#computers
```

Listing 53: Special Case

```
#On Python Shell
'Python'[ : : -1 ]
#Output: 'nohtyP'

'Python'[ 10 : -10 ]
#Output:' ' (empty string)
```

Tuples I

- Tuple: Consists of a number of values separated by commas

Listing 54: Tuples, emply and singleton tuples

```
t = ("intro to tuples", "ami choksi", 20)
for i in range(0,3):
        print(t[i])
print(t)
print(type(t))
c=()#Empty tuple
print(c)
singleton = (1,)#singleton tuple, at the end ’,’ is must
print(singleton)
’’’Output
intro to tuples
ami choksi
20
(’intro to tuples’, ’ami choksi’, 20)
```

Tuples II

```
<class ’tuple’>
()
(1,)
’’’
```

- Nested Tuple

Listing 55: Nested Tuple

```
course = ("Python", "Dr.Ami Choksi", 20)
student = ("Ishu", 20, course)
print(student)#(’Ishu’, 20, (’Python’, ’Dr.Ami Choksi’, 20))

#length of the Tuple
print("course length",len(course))#3
print("student length",len(student))#3
empty=()
print("Empty length",len(empty))#0
```

Tuples III

```
singleton=(1,)
print(singleton, len(singleton))#1

'''Output:
('Ishu', 20, ('Python', 'Dr.Ami Choksi', 20))
course length 3
student length 3
Empty length 0
(1,) 1
'''
```

- More operations on Tuples

Tuples IV

Listing 56: "More Operations on Tuples

```
course1=("Py","ami",20)
course2=("Dm","ac",30)
print("Concat",course1+course2)#('Py', 'ami', 20, 'Dm', 'ac', 30)
print("multiply",2*course2)#('Dm', 'ac', 30, 'Dm', 'ac', 30)
print("multiply",course2*2)#('Dm', 'ac', 30, 'Dm', 'ac', 30)

'''Ouptut:
Concat ('Py', 'ami', 20, 'Dm', 'ac', 30)
multiply ('Dm', 'ac', 30, 'Dm', 'ac', 30)
multiply ('Dm', 'ac', 30, 'Dm', 'ac', 30)
'''
```

Listing 57: Special case

```
What is the value of the expression (using python 3.X version):
len(1,2,3,4,5)
#Output: ERROR - len() takes exactly one argument
```

Tuples V

- Unpacking sequences
- Strings and tuples are examples of sequences
- Indexing,slicing, concatenation, repetition operations are applicable on seqences
- It is used to have multiple assignment
- LHS and RHS must have equal length

Listing 58: Unpacking sequences

```python
course=("Python","ami",22)
student=("ibhu",10,course)
print("student : ",student)

name, roll, regcourse=student

print("name : ",name)
print("roll : ",roll)
```

Tuples VI

```
print("regcourse : ",regcourse)

x1,x2,x3,x4 = ’amic’
print(x1)#a
print(x4)#c

#if we give x1,x2,x3,x4="ami"
#ValueError: not enough values to unpack (expected 4, got 3)

#x1,x2="ami"
#too many values to unpack (expected 2)
```

Tuples VII

Listing 59: Unpacking sequence:1 more example

```
(x,y) = (1,(2,4))[1]
print(y)#4

(x,y) = (1,
print(y)#ValueError: not enough values to unpack (expected 2, got 1
#cause
#(1,(2,4))[1:] #Output ((2, 4),)
#(x,y)=((2,4),)
#y doesn't get any value

(x,y) = (1,(2,4))
print(y)#(2,4)


(x, y) = ("Hello","AK")
print(y)#AK
```

Tuples VIII

```python


(x, (y1, y2)) = ("Hello","AK")
print(y1)#A
```

Listing 60: Pass Tuple to function

```python
#Pass tuple to function
def printTupleElements(t):
       for i in t:
              print(i)

tp=(1,2,3,4,5,6)
printTupleElements(tp)
#Output:
#1
#2
#3
```

Tuples IX

```
#4
#5
#6
```

Listing 61: Pass two tuple to function

```
#Pass two tuple to function
def printTupleElements(t1,t2):
       for i in t1:
              print(i)
       for i in t2:
              print(i)

printTupleElements((1,2),(4,5))
#Output:
#1
#2
#4
```

Tuples X

```
#5
```

List I

- Ordered sequence of values
- Comma separated values in square brackets
- Values can be of different types, usually of same types
- Sequence operations are applicable to lists
- len, indexing, slicing, concatenation, repeatition, unpacking are applied to lists
- More Operations
- append(x), extend(seq), insert(i,x), remove(x), pop(i), pop(), index(x), count(x), sort, reverse()
- index() and count() will not change the list, Other operations in above point will change the list

List II

- The key difference is that tuples are *immutable* . This means that you cannot change the values in a tuple once you have created it. As a list is *mutable* , it can’t be used as a key in a dictionary, whereas a tuple can be used.
- Lists are for variable length , tuples are for fixed length .
- The literal syntax of tuples is shown by parentheses () whereas the literal syntax of lists is shown by square brackets [] .
- Tuples are heterogeneous data structures (i.e., their entries have different meanings), while lists are homogeneous sequences.
- Tuples show structure whereas lists.
- Tuples have O(N) append, insert, and delete performance whereas Lists have O(1) append, insert and delete performance.

List III

Listing 62: Python shell

```
#On Python shell
>>> list=[1,2,3,4,5]
>>> type(list)
<class ’list’>
```

Listing 63: List Operations

```
# x and w are same: to see the difference between append and extend
x=[1,2,3,4,"five"]
w=[1,2,3,4,"five"]
# y and z are same: to see the difference between append and extend
y=["six","seven","eight",9,10]
z=["six","seven","eight",9,10]

x.append(y)
print(x)#[1, 2, 3, 4, ’five’, [’six’, ’seven’, ’eight’, 9, 10]]

```

List IV

```python
w.extend(y)
print(w)#[1, 2, 3, 4, 'five', 'six', 'seven', 'eight', 9, 10]

y.extend(y)
print(y)#['six', 'seven', 'eight', 9, 10, 'six', 'seven', 'eight',


# x.append will append to x but it doesn't return anything,
# so printing that returns None}
x=[1,2,3,4,"five"]
y=["six","seven","eight",9,10]
print(x.append(y))#None
```

List V

Listing 64: find the names from list whose last letter is ’u’

```
#find the names from list whose last letter is ’u’
names=[’pinku’,’tinku’,’chinku’,’tonny’,’sony’]
for i in names:
        if i[-1]==’u’:
                print(i,end=".")#pinku.tinku.chinku.
```

Listing 65: join()

```
#making a string out of list and string to list
#list to string: join
l=[’my’,’name’,’is’,’ami’]
print(l)#[’my’, ’name’, ’is’, ’ami’]
b=’’.join(l)
print(b)#’mynameisami’
```

List VI

Listing 66: More List Operations

```
'''\begin{itemize}
\item Ordered sequence of values
\item Comma separated values in square brackets
\item Values can be of different types, usually of same types
\item Sequence operations are applicable to lists
\item len, indexing, slicing, concatenation, repeatition, unpacking
'''
x = [1,2,3,4,"five"]
print(x[0])#1
print(x[4])#five
print(len(x))#5
print(x[-1])#five
print(x[-20:20])#[1, 2, 3, 4, 'five']
print(x[-20:4]) #[1, 2, 3, 4]
y=["six","seven","eight",9,10]
z=x+y
```

List VII

```python
print(z)#[1, 2, 3, 4, 'five', 'six', 'seven', 'eight', 9, 10]

z=2*x
print(z)#[1, 2, 3, 4, 'five', 1, 2, 3, 4, 'five']

#x1,x2,x3,x4 = x#ValueError: too many values to unpack (expected 4)
#x1,x2,x3,x4,x5 = x;

x=[5,4,3,2,1]
print(x)#[5, 4, 3, 2, 1]
```

List VIII

Listing 67: Pass List to a function

```python
#Pass Lists to a function
#Pass Lists to a function
def printList(ll1,ll2):
       for i in ll1:
              print(i)
       for i in ll2:
              print(i)

printList([1,2],[3,4])
#Output:
#1
#2
#3
#4
```

Mutable and immutable types I

- Tuples and lists look very similar.
- Tuples and Strings are immutable.
- Lists are mutable.

Listing 68: Mutable List and Unmutable Tuple

```
indoor=[’badminton’,’table tennis’, ’carrom’]
outdoor=[’football’,’cricket’]
games=(indoor, outdoor)
print("games : ",games)
#games : ([’badminton’, ’table tennis’, ’carrom’], [’football’,


cards=[’sol’,’hearts’,’freecell’]
print("games[0] : ",games[0])
#games[0] : [’badminton’, ’table tennis’, ’carrom’]
#games[0] = cards# ’tuple’ object does not support item assignm
games1=(indoor, outdoor, cards)
```

Mutable and immutable types II

```
print(games1)
#([’badminton’, ’table tennis’, ’carrom’], [’football’, ’cricke
#Output: games : ([’badminton’, ’table tennis’, ’carrom’], [’fo
#games[0] : [’badminton’, ’table tennis’, ’carrom’]
([’badminton’, ’table tennis’, ’carrom’], [’football’, ’cricket
```

Listing 69: List Slicing

```
games1=([’badminton’, ’table tennis’, ’carrom’], [’football’, ’
print(games1[:2])
#([’badminton’, ’table tennis’, ’carrom’], [’football’, ’cricke

indoor=[’badminton’,’table tennis’, ’carrom’]
outdoor=[’football’,’cricket’]
games =(indoor, outdoor)

print("games " , games)
#games ([’badminton’, ’table tennis’, ’carrom’], [’football’,
```

Mutable and immutable types III

```

#replaces outdoor[0]
outdoor[0]=’hockey’
#([’badminton’, ’table tennis’, ’carrom’], [’hockey’, ’cricket’
if games1[:2] == games:
       print("True")
else:
       print("False")

#False
```

- Tuple *games* is having reference of *outdoor*, so when outdoor is getting changed, games too is getting changed.
- e in seq: True if *e* is there in *seq*

Mutable and immutable types IV

Listing 70: Change to list will change tuple which reference it

```
indoor=(’badminton’, ’table tennis’, ’carrom’)
games=([’badminton’, ’table tennis’, ’carrom’], [’football’, ’c
if ’badminton’ in indoor:
       print(True)
else:
       print(False)
#True
#in if condition, indoor is replaced with games, it returns Fal
```

- e not in seq:True if *e* is not there in *seq*
- for e in seq:Iterate over all elements in *seq*

Mutable and immutable types V

```
indoor=(’badminton’, ’table tennis’, ’carrom’)
games=([’badminton’, ’table tennis’, ’carrom’], [’football’, ’c
for i in games:
       print (i)

#Output: [’badminton’, ’table tennis’, ’carrom’]
#[’football’, ’cricket’]
```

Listing 71: append()

```
v1 = [1,2,3]
t1 = (v1,v1[:])
v1.append(4)
print (t1)#([1,2,3,4],[1,2,3])
```

- Steps to change an individual element of tuple :
    - Convert the tuple to list.
    - Change the element in the list.

Mutable and immutable types VI

- Convert changed the list back to tuple.

Listing 72: Change immutable tuple

```python
#Program i submitted during learning of NEAT-AICTE Python cours
mytuple = ("i", "love", "python")
print("Given Tuple:",mytuple)
list1 = list(mytuple)
print("After Converting Tuple into List:",list1)
list1[1]="practice"
print("List after changing element:",list1)
mytuple=tuple(list1)
print("After Converting List into Tuple:",mytuple)
’’’
Output:
Given Tuple: (’i’, ’love’, ’python’)
After Converting Tuple into List: [’i’, ’love’, ’python’]
List after changing element: [’i’, ’practice’, ’python’]
```

Mutable and immutable types VII

```
After Converting List into Tuple: (’i’, ’practice’, ’python’)
’’’
```

List Comprehension I

- A concise way to build a list
- It consists of square brackets containing an expression followed by a `for` clause, then zero or more `for or if` clauses. The expressions can be anything, meaning you can put in all kinds of objects in lists.
- The result will be a new list resulting from evaluating the expression in the context of the `for and if clauses` which follow it.
- The list comprehension always returns a result list.

Listing 73: Simple way of creating list

```python
#Simple way of creating list
old_list=[1,2,3,4,5,6]
new_list = []
for i in old_list:
        new_list.append(i+2)
print(new_list)#[3, 4, 5, 6, 7, 8]
```

List Comprehension II

Listing 74: Equivalent List Comprehension

```python
#List comprehension of above example
new_list=[x+2 for x in old_list]
print(new_list)#[3, 4, 5, 6, 7, 8]
```

Listing 75: List comprehension example

```python
y=[x*x for x in range(1,11)]
print(y)
#[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]


nums=[-1,25,9,-30,5]
pos = [x for x in nums if x>0]
print(pos)#[25, 9, 5]


neg = [x for x in nums if x<0]
```

List Comprehension III

```python
print(neg)#[-1, -30]

absp = [(x, abs(x)) for x in nums]
print(absp)
#[(-1, 1), (25, 25), (9, 9), (-30, 30), (5, 5)]

sqlist=[(x,y) for x in [1,2,3] for y in [12,9,27,4,50] if x*x ==y]
print(sqlist)#[(2, 4), (3, 9)]
```

```python
# how if adding 3 separate parameters the result
# is confusing. How is this returning these
# particular results? (4 6 and 8) ????

for i in range(4, 10, 2):
       print(i, end=" ")
#4 6 8
#range(start, stop[, step])

```

List Comprehension IV

```
#Starts at 4, then increments by 2, to end at 8 because 10 < 10 is
```

Listing 76: for example

```
for y in range(3,1,-1):
        print(y)
#(3,2)
#Starts at 3, then increments by -1, so 2, then again increments by
```

Listing 77: Comprehension with %

```
z = [x*x for x in range(1,5) if (x%2 == 0)]
print(z)#[4, 16]
```

Listing 78: Comprehension with range

```
z = [x*y for y in range(3, 1, -1) for x in range(1,4) ]
print(z)#[3,6,9,2,4,6]
```

Generators I

- Generator Expressions are somewhat similar to list comprehensions, it does not construct list object.
- Instead of creating a list and keeping the whole sequence in the memory, the generator generates the next element in demand.
- When a normal function with a return statement is called, it terminates whenever it gets a return statement.
- But a function with a yield statement saves the state of the function and can be picked up from the same state, next time the function is called.
- The Generator Expression allows us to create a generator without the yield keyword.
- **Syntax Difference:** Parenthesis are used in place of square brackets.

Generators II

Listing 79: List Comprehension Example

```python
#List Comprehension Example
list_comprehension = [i for i in range(11) if i % 2 == 0]
print(list_comprehension)
#[0, 2, 4, 6, 8, 10]
```

Listing 80: Generators Example

```python

# Generator Expression
generator_expression = (i for i in range(11) if i % 2 == 0)
print(generator_expression)
#<generator object <genexpr> at 0x0000000004FC41B0>

#Iterate over Generator
for i in generator_expression:
       print(i, end=" ")#0 2 4 6 8 10
```

Generators III

Listing 81: Get size of list comprehension and generator

```python
from sys import getsizeof

comp = [i for i in range(10000)]
gen = (i for i in range(10000))

#gives size for list comprehension
x = getsizeof(comp)
print("x = ", x)

#gives size for generator expression
y = getsizeof(gen)
print("y = ", y)
#x = 87624
#y = 120
```

Sets I

- Another type of sequence objects.
- It is unordered data set.
- It eliminates duplicate entries
- Union, intersection, difference are the operations.

Listing 82: Use of Set

```
#set is defined as
s={1,2,3,3,4}
#it ONLY considers unique elements
print(s)#{1, 2, 3, 4}
```

Sets II

Listing 83: Use of in Operator

```
basket=[’apple’,’banana’,’apple’,’pear’,’orange’,’banana’]
fruits=set(basket)
print("fruits: ",fruits)#fruits: {’orange’, ’pear’, ’banana’, ’appl
print("Type: ",type(fruits))#Type: <class ’set’>

if ’apple’ in fruits:
       print("True")#True
else:
       print("False")


if ’mango’ in fruits:
       print("True")
else:
       print("False")#False

```

Sets III

```
#Output
#fruits: {’pear’, ’banana’, ’apple’, ’orange’}
#Type: <class ’set’>
#True
#False
```

Listing 84: Set Operations

```
A=set(’acads’)
B=set(’work’)
print("A: ",A)
print("B: ",B)
print("A-B:",A-B)
print("AUB : ", A|B)
print("A&B",A&B)
#symmetric difference
print("A^B: ", A^B )#AUB - A&B
'''Output:
```

Sets IV

```
A: {’a’, ’s’, ’c’, ’d’}
B: {’u’, ’s’, ’t’, ’n’, ’i’, ’e’}
A-B: {’a’, ’c’, ’d’}
AUB : {’u’, ’i’, ’c’, ’a’, ’s’, ’t’, ’n’, ’d’, ’e’}
A&B {’s’}
A^B: {’u’, ’a’, ’c’, ’d’, ’t’, ’n’, ’i’, ’e’}
’’’
```

Listing 85: Duplicate removal from a list

```python
#Duplicate removal
list1 = [1,2,3,5,3,2,7]
print(list1)#[1,2,3,5,3,2,7]
s=set(list1)
list1=list(s)
print("After removing duplicates",list1)#[1, 2, 3, 5, 7]
’’’Output:
[1, 2, 3, 5, 3, 2, 7]
```

Sets V

```
After removing duplicates [1, 2, 3, 5, 7]
’’’
```

Listing 86: Special Cases

```
snacks = set([’burger’, ’fries’, ’pizza’ , ’fries’, ’toast’,
’peanuts’, ’fries’, ’pizza’])
print((’fries’ in snacks) > (’pizza’ in snacks))#False
```

Listing 87: Explanation of above example

```
print(’fries’ in snacks)#True
print(’pizza’ in snacks)#True
print(True>True)#False
```

Sets VI

Listing 88: Difference of set of strings

```
s1 = set([’burger’, ’fries’, ’pizza’ , ’fries’,
 ’toast’, ’peanuts’, ’fries’, ’pizza’])
s2 = set([’burger’, ’fries’, ’burger’, ’fries’,
 ’omlette’])
print(s1-s2)#{’toast’, ’peanuts’, ’pizza’}
```

Listing 89: Pass set to a function

```
#Pass set to a function
def printSet(s):
       for i in s:
              print(i)

printSet({1,2,3})
#Output:
#1
#2
```

Sets VII

```
#3
```

Dictionary I

Feature	List	Dictionary
Ordered/ Unordered	Ordered. They maintain the order in which elements are inserted.	Unordered. Ordering is not guaranteed here.

How assigned	Comma separated values in [].
	list1=[1, 2, 3, 4, 5, 6]
	list2=["ami","ash"]
		Comma separated key:value in {}.
		dict1={"name":"Ami",
		"nationality":"Indian"}
		dict2={0:"ami",1:"ash"}

Access using	It is accessed using index. e.g.list1[0], list2[1]	It is accessed using key. e.g.dict1["name"], dict2[0]
When to Use	It should be used when an ordered collection of items is required.	It should be used when a set of unique keys that map to values and to use.

Print	print(list1)
	Or
	print(list1[0:])
	output: [1, 2, 3, 4, 5, 6]
		print(dict1)
		output: {'name': 'Ash',
		'nationality': 'Indian'}

Dictionary II

Iterate using	(i)for i in range(len(list1)):
	print(list1[i])
		(i)for i in dict1.keys():
		print(i,":",dict1[i])
		(ii)for key,val in dict1.items():
		print(key, "=>", val)

Indexing	Lists have positive and negative list indexing.	Keys are only used as indexes.
Remove item	#1 will be removed from list list1.remove(1)#1 will be removed list1.pop(1)#1st index item, i.e. 2 will be deleted	#name and its value is removed from dict1 del dict1['name'] dict1.pop('name')
Sorting	list1.sort()	Sort keys: sorted(dict1.keys()) Sort values: sorted(dict1.values())

Dictionary III

- It is unordered set of key: value pairs, with the requirement that the keys are unique and immutable.
- keys() and values() are the methods to extract keys and values of the dictionary object.
- key:value pairs are in {}

Listing 90: keys and values list

```
dict1 = {’name’:’ami’,’des’:’Associate Prof’, ’work’:’ckpcet’}
print("Dict : ",dict1)
#Dict : {’name’: ’ami’, ’des’: ’Associate Prof’, ’work’: ’ckpcet’}

print(dict1[’name’])#ami
print(dict1.keys())
#dict_keys([’name’, ’des’, ’work’])
print(dict1.values())
#dict_values([’ami’, ’Associate Prof’, ’ckpcet’]
```

Dictionary IV

- If one has provided two values for the same key, 2nd value overrides the first one.

Listing 91: Two values of same key provided

```
dict1 = {’name’:’ami’,’des’:’Associate Prof’,
’work’:’ckpcet’, ’name’:’abhay’}
print("Dict : ",dict1)


print(dict1[’name’])#abhay
print(dict1.keys())
#dict_keys([’name’, ’des’, ’work’])
print(dict1.values())
#dict_values([’abhay’, ’Associate Prof’, ’ckpcet’])
’’’Output:
Dict : {’name’: ’abhay’, ’des’: ’Associate
Prof’, ’work’: ’ckpcet’}
```

Dictionary V

```
abhay
dict_keys(['name', 'des', 'work'])
dict_values(['abhay', 'Associate Prof', 'ckpcet'])
'''
```

- Empty dictionary is created by writing {}
- Dictionaries are mutable: add new key:value, changed pairing, delete key

Listing 92: Empty dictionary creation

```python
#empty dictionary creation
dict1={}
print(dict1)#{}
print(type(dict1))#<class 'dict'>
```

Dictionary VI

Listing 93: Dictionary Operations

```
dict1 = {’name’:’ami’,’des’:’Associate Prof’,
 ’work’:’ckpcet’, ’name’:’abhay’}
#length
print("len : ", len(dict1))#3

#keys
print("keys: ", dict1.keys())
#keys: dict1_keys([’name’, ’des’, ’work’])

#values
print("values:",dict1.values())
#values: dict1_values([’abhay’, ’Associate Prof’, ’ckpcet’])

#key available
print(’name’ in dict1)#True
print("key available:",dict1.get(’nam’,’ruchi’))#ruchi
```

Dictionary VII

```
#if key is not available it throws exception, to solve it
#with get method, default value is given

#key not available
print("key not available:",dict1.get(’name’,’ruchi’))#abhay
dict1[’name’]=’chamcham’
print(dict1[’name’])#chamcham

#delete particular key
del dict1[’name’]
print("dict1:",dict1)
#dict1: {’des’: ’Associate Prof’, ’work’: ’ckpcet’}

#iterate dictionary
for i in dict1:
       print("Key : ", i," Value : ",dict1[i])
#Output for loop
#Key : des Value : Associate Prof
```

Dictionary VIII

```
#Key : work Value : ckpcet

’’’Output:
len : 3
keys: dict_keys([’name’, ’des’, ’work’])
values: dict_values([’abhay’, ’Associate Prof’, ’ckpcet’])
True
key available: ruchi
key not available: abhay
chamcham
dict: {’des’: ’Associate Prof’, ’work’: ’ckpcet’}
Key : des Value : Associate Prof
Key : work Value : ckpcet’’’
```

Dictionary IX

Listing 94: del dict required

```
#if the code gives error of "TypeError: 'dict' object is not
#callable", do
#del dict first then it works fine
del dict
airport=dict([('mum','bom'),('chen','maa')])
print(airport)#{'mum': 'bom', 'chen': 'maa'}

bob=dict(name='bob smith',age=42,pay='10000',job='dev')
print(bob)
#{'name': 'bob smith', 'age': 42, 'pay': '10000', 'job': 'dev'}
```

Dictionary X

Listing 95: Sorting dictinary

```
#sorting dictinary
key_value ={}
key_value[2] = 56
key_value[1] = 2
key_value[5] = 12
key_value[4] = 24
key_value[6] = 18
key_value[3] = 323
#sorting dictinary with keys
print(sorted (key_value.keys()))
#sorting dictinary with values
print(sorted(key_value.values()))
'''
Output:
[1, 2, 3, 4, 5, 6]
[2, 12, 18, 24, 56, 323]
```

Dictionary XI

```
'''
```

Listing 96: Pass dictionary to a function

```
#Pass dictionary to a function
def printDictionary(dc):
       for i in dc.keys():
              print(i,":",dc[i])

dct={1:11,2:22,3:33}
printDictionary(dct)
#Output:
#1 : 11
#2 : 22
#3 : 33
```

Higher order functions I

- In Python, functions are first-class objects. There is no distinction between data and functions.
- Functions have types
- Functions can be passed as arguments
- Functions can be returned.
- Functions can be members of sequences

Listing 97: Pass Function as Argument

```python
def summation(n,f):
       sum=0
       for i in range(1,n+1):
              sum=sum+f(i)
       return sum

def identity(x):
       return x
```

Higher order functions II

```python

def square(x):
       return x*x

def cube(x):
       return x**3

print("sum: " ,summation(10,identity))#sum: 55
print("square: ", summation(10,square ))#square: 385
print("cube: ", summation(10, cube))#3025

'''Output:
sum: 55
square: 385
cube: 3025 '''
```

- Lamda Function

Higher order functions III

- To create anonymous functions
- Syntaxlambda param1, param2, paramk:Body

Listing 98: Lambda function

```
#Lambda function
def summation(n,f):
        sum=0
        for i in range(1,n+1):
                sum=sum+f(i)
        return sum

print(summation(10,lambda x:x))#55
print(summation(10,lambda x:x*x))#385
print(summation(10,lambda x:x*x*x))#3025

'''Output
55
```

Higher order functions IV

```
385
3025 '''
```

Listing 99: Special Case

```
def printapp(f, xs):
        for i in xs:
                print(f(i), end=’%’)

def double(x):
        return x+x

printapp(double, [1,2,3])
#Output: 2%4%6%
```

Higher order functions V

```python
def printapp(f, xs):
       for i in xs:
              print(f(i), end='')

printapp(lambda x: x+x, "str")
#OUtput: ssttrr
```

Listing 100: Special Case

```python
w = (lambda xs,ys:list(x+y for x in xs for y in ys))((1,2),(9, 11))
print (w)
#Output: [10, 12, 11, 13]
```

Higher order functions VI

Listing 101: Special Case

```python
def printapp(f, xs):
        for i in xs:
                print(f(i), end=’’)

printapp(lambda x: x+x, [(1,2), (3,4)])
#(1, 2, 1, 2)(3, 4, 3, 4)
```

Listing 102: Special Case

```python
def printapp(f, xs):
        for i in xs:
                print(f(i), end=’%’)

def double(x):
        return x+x
printapp(double, {1:1, 2:4, 3:9})

```

Higher order functions VII

```
#Output: 2%4%6%
```

Sieve Eratosthenes's Prime number algorithm I

- Remove 0 and 1, as they are composite numbers
- Strike out, set false for that number in array, 2 and its multiples for range(2*j,n+1)
- Repeat the same procedure for numbers till n, strike out multiples of number

Listing 103: Sieve Eratosthenes's Algorithm

```
"""Generating prime numbers
Created on Fri Jul 20 13:40:11 2018
Implementation of Sieve Eratosthenes's Prime number finding
algorithm
from IITK Python-practical programming, by Prof. Amey Karkare
@author: Dr. Ami Tusharkant Choksi
"""
def sieve(n):
       global primes
```

Sieve Eratosthenes’s Prime number algorithm II

```
        primes=[True]*(n+1)
        #print(primes)
        primes[0], primes[1]=False, False;
        #0 & 1 are treated composite

        for j in range(2, n+1):
                if primes[j]==False:
                        continue
                for i in range(2*j, n+1, j):
                        primes[i] = False

global primes
n = int(input(’n= ’))

sieve(n)
for i in range(2, n+1):
        if primes[i]:
                print(i, end=’ ’)
```

Sieve Eratosthenes’s Prime number algorithm III

```

#Prime No.s 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
# 71 73 79 83 89 97
```

File I/O I

- Files are persistent storage.
- Operations: open, close, read, write
- Python treat files as sequence of files, so sequence operations work for the data from files
- File I/O: Open and Close
- open(filename, mode)
- mode: r-read, w-write, a-append
- for ’w and ’a’, if file is not available, it creates the file. The file is created in the same directory where in our the program is running.

Access Modes	Description
r	Opens a file for reading only.
rb	Opens a file for reading only in binary format.
r+	Opens a file for both reading and writing.

File I/O II

rb+	Opens a file for both reading and writing in binary format.
w	Opens a file for writing only.
wb	Opens a file for writing only in binary format.
w+	Opens a file for both writing and reading.
wb+	Opens a file for both writing and reading in binary format.
a	Opens a file for appending.
ab	Opens a file for appending in binary format.
a+	Opens a file for both appending and reading.
ab+	Opens a file for both appending and reading in binary format.

File I/O III

Listing 104: File IO write

```
#open(filename, mode)
test=open("test.txt","w")
test.writelines("hello and welcome to the world of computers")
test.write("\nhello friends")
test.write("\nRushi, ishu")
test.close()
'''Content of test.txt
hello and welcome to the world of computers
hello friends
Rushi, ishu
'''
```

File I/O IV

Listing 105: File IO read

```
test=open("test.txt","r")
for line in test:
        print(line,end='')
test.close()
print("\ntest Obj",test)
'''Output:
hello and welcome to the world of computers
hello friends
Rushi, ishu
test Obj <_io.TextIOWrapper name='test.txt' mode='r'
encoding='cp1252'>
'''
```

Listing 106: tennisPlayers.txt

```
stefi
martina
```

File I/O V

Listing 107: tennisCountries.txt

```
India
England
```

Listing 108: Sequence Operations on Content of File

```
fn=open('tennisPlayers.txt','r')


pn, pc = [], []
for i in fn:
        pn.append(i[:-1])#ignore \n
fn.close()

fc = open('tennisCountries.txt','r')
for i in fc:
        pc.append(i[:-1])
fc.close()
```

File I/O VI

```python
print(pn,'\n',pc)


'''Output:
['stefi', 'martina']
['India', 'England'] '''
```

Listing 109: Making Dictionary out of Files Contents

```python
fn=open('tennisPlayers.txt','r')


pn, pc = [], []
for i in fn:
        pn.append(i[:-1])#ignore \n
fn.close()

fc = open('tennisCountries.txt','r')
```

File I/O VII

```python
for i in fc:
        pc.append(i[:-1])
fc.close()
print(pn,'\n',pc)


print("len",len(pc))
nameCountry = []
for i in range(len(pc)):
        nameCountry.append((pn[i],pc[i]))
n2c = dict(nameCountry)
print(n2c)



'''Output:
['stefi', 'martina']
['India', 'England']
```

File I/O VIII

```
len 2
{'stefi': 'India', 'martina': 'England'} '''
```

- theFile.seek(pos, ref) - modify the file object for theFile so that the next read will be from

Listing 110: Contents of seek1.txt

```
first line
second line
third line
fourth line
fifth line
```

File I/O IX

Listing 111: Seek operation

```
# seek operation
f = open("seek1.txt", "r")

f.seek(4)
for line in f:
       print(line,end='')
f.close()
'''
Output:
t line
second line
third line
fourth line
fifth line
'''
```

File I/O X

Listing 112: Append operation

```
fo = open('seek1.txt', 'a')

# Append 'hello' at the end of file
fo.write('hello')

# Close the file
fo.close()
'''
Contents of seek1.txt after append operation
first line
second line
third line
fourth line
fifth linehello
'''
```

Exceptions I

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal:
- e.g.`10/0` causes `ZeroDivisionError:  division by zero`
- Exceptions come in different types, and the type is printed as part of the message: the types in the example are ZeroDivisionError, NameError and TypeError

Listing 113: Simple exception handling

```python
while True:
       try:
              x = int(input("Please enter a number: "))
              break
       except ValueError:
              print("Oops! That was no valid number. Try again...")
```

Exceptions II

```
'''
Output:
Please enter a number: 5.5
Oops! That was no valid number. Try again...
Please enter a number: 5
'''
```

- Code in try if raises exception is being handled by the except block.
- The try statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

Exceptions III

Listing 114: Raise the exception and Finally

```python
try:
        raise KeyboardInterrupt
finally:
        print('Goodbye, world!')


'''Output:
Goodbye, world!
Traceback (most recent call last):

File "<ipython-input-6-a9f4316f68bc>", line 2, in <module>
raise KeyboardInterrupt

KeyboardInterrupt
'''
```

Exceptions IV

Listing 115: IOError Exception

```
#If directory is not writable IOError Exception
try:
        #/target.txt generated IOError
        #if changed to target.txt no exception is generated
        #w=open(’/target.txt’,’w’)
        w=open(’target.txt’,’w’)
except IOError:
        print("Can not write to the target file")
else:
        w.write(’success’)
        print(’Can write in dir’)
w.close()

’’’
#/target.txt
Can not write to the target file
```

Exceptions V

```
#target.txt
Can write in dir
'''
```

Listing 116: Try, except, else, finally

```
try:
        x = int(input("x = "))
        y = int(input("y = "))
        result = x/y
except ValueError:
        print("Bad number in input")
except ZeroDivisionError:
        print("division by zero")
else:
        print("result is", result)
finally:
        print("program finished")
```

Exceptions VI

```

'''Output:
x = 5
y = 0
division by zero
program finished'''

'''Output:
x = 5
y = 5.6
Bad number in input
program finished '''

'''Output:
x = 7
y = 5
result is 1.4
program finished '''
```

Exceptions VII

Listing 117: Special Case

```
try:
        print(’Hello’, end=’#’)
except:
        print(’Bye’, end=’#’)
else:
        print(’PK’, end=’#’)
finally:
        print(’AK’, end=’#’)

#Output: Hello#PK#AK#
```

Exceptions VIII

Listing 118: Special Case

```
try:
       5/0
except:
       print("Something Bad Happened")
except ZeroDivisionError:
       print("Division by Zero")
except ArithmeticError:
       print("Bad Arithmetic")
#Output: 5/0: SyntaxError: default ’except:’ must be last
```

Exceptions IX

Listing 119: Modified above

```python
try:
       5/0
except ZeroDivisionError:
       print("Division by Zero")
except ArithmeticError:
       print("Bad Arithmetic")
except:
       print("Something Bad Happened")
#Output:Division by Zero
```

Exceptions X

Listing 120: Assert example

```python
x = "hello"
#if condition returns True, then nothing happens:
try:
       assert x == "hello"
except AssertionError:
       print("Assert error hello")
else:
       print("ok hello")
#if condition returns False, AssertionError is raised:
try:
       assert x == "goodbye"
except AssertionError:
       print("Assert error goodbye")
else:
       print("ok goodbye")

```

Exceptions XI

```
# Output:
#ok hello
#Assert error goodbye
```

- If you want to store the result of while condition and want to proceed to,
- e.g.`while(()a=functioncall())!=0)` in C/C++/Java like languages, we need to import let library.

Listing 121: While:Result of Condtition Storing

```python
from let import let
while let(closeDoorList=getCloseDoorList()>1):
  choice=int(input("Enter your door choice {}"
  .format(closeDoorList,":")))
```

print odd even char Example I

Listing 122: Print Odd indexed and even indexed elements separated by space

```
'''
Problem Statement: Given a string,S, of length N that is indexed
from 0 to N-1 , print its even-indexed and odd-indexed characters
as 2 space-separated strings on a single line.

Sample Input:
Hacker
Rank

Sample Output:
Hce akr
Rn ak
'''
s=input("Enter a string : ")
```

print odd even char Example II

```
listt=list(s)
n=len(listt)
lOdd=''.join(listt[1::2])
lEven=''.join(listt[0::2])
print("Output: ",lEven,lOdd)

'''
#Output:
Enter a string : hacker
Output: hce akr
hce akr
Enter a string rank
Output: rn ak
'''
```

References I

1. Python tutorial, https://www.tutorialspoint.com/python/python_overview.htm
2. The Python Tutorial, https://docs.python.org/3/tutorial/
3. Dr.Jeeva Jose, Taming Python by Programming book
4. David Mertz, Picking a Python Version: A Manifesto book, http://www.oreilly.com/programming/free/files/from-future-import-python.pdf
5. Numpy, https://campus.datacamp.com/courses/intro-to-python-for-data-science/chapter-4-numpy?ex=2
6. Standard data types, https://www.tutorialspoint.com/python3/python_variable_types.htm
7. Numbers, https://docs.python.org/3/tutorial/introduction.html#numbers

References II

8. Strings, https://docs.python.org/3/tutorial/introduction.html#numbers
9. Python, Practical programming approach, IIT Kanpur course By Amey Karkare, Department of CSE
10. Print formatting, https://www.python-course.eu/python3_formatted_output.php
11. If else in python, https://www.tutorialspoint.com/python/python_if_else.htm
12. Operator Precedence in Python, https://docs.python.org/3/reference/expressions.html?highlight=operator%20precedence#operator-precedence
13. While else, https://docs.python.org/3/reference/compound_stmts.html
14. Global, https://stackoverflow.com/questions/10506973/can-not-increment-global-variable-from-function-in-python

References III

15. Lamda-anonymous function, `https://www.python-course.eu/lambda.php`
16. String functions, `https://www.programiz.com/python-programming/methods/string`
17. boolean, `https://problemsolvingwithpython.com/04-Data-Types-and-Variables/04.02-Boolean-Data-Type/`
18. Os module, `https://www.tutorialspoint.com/python/os_file_methods.htm`
19. Constructor and destructor, `https://helloacm.com/constructor-and-destructor-in-python-classes/`
20. Email collecting script, `https://gist.github.com/dhruvbaldawa/1476680`
21. Difference between tuple and list, `http://net-informations.com/python/iq/tup.htm`

References IV

22. List Comprehension, https://stackoverflow.com/questions/32096391/pythons-range-function-with-3-parameters
23. Generators, https://www.geeksforgeeks.org/python-list-comprehensions-vs-generator-expressions/
24. File reading, https://www.pythonforbeginners.com/cheatsheet/python-file-handling
25. Access modes of file open, https://www.tutorialsteacher.com/python/python-read-write-file
26. assert keyword, https://www.w3schools.com/python/ref_keyword_assert.asp