BansilalRamnathAgrawal Charitable Trust's

VISHWAKARMA INSTITUTE OF INFORMATION TECHNOLOGY

Department of Engineering &Applied Sciences

F. Y. B. Tech.

Course Material (A Brief Reference Version for Students)

Course: Python for Engineers

Unit I – Introduction to python

Unit I – Introduction to python

Script Model Programming, Understanding Python variables, basic Operators, Declaring and using Numeric data types: int, float, complex, Using string data type and string operations, Defining list and list slicing, List manipulation using in build methods, Use of Tuple data type , Dictionary manipulation .

A. Python (Script Model Programming)

- Python files have the extension .**py**
- Python is a high-level, interpreted, interactive and object-oriented scripting language.
- Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.
- **Python is Interpreted**: Python is processed at runtime by the interpreter. You do not need to compile your program before executing it.
- **Python is Interactive**: You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.

Features of Python

1. **Easy-to-learn**:
   Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
2. **Easy-to-read:**
   Python code is more clearly defined and visible to the eyes.
3. **Easy-to-maintain:**
   Python's source code is fairly easy-to-maintain.
4. **A broad standard library:**
   Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
5. **Interactive Mode:**
   Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
6. **Portable:**
   Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
7. **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.
8. **Databases:**
   Python provides interfaces to all major commercial databases.
9. **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.
10. **Scalable:**
    Python provides a better structure and support for large programs than shell scripting

B.Python variables

One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value. Programmers generally choose names for their variables that are meaningful and document what the variable is used for

Python is not "statically typed". We do not need to declare variables before using them, or declare their type. A variable is created the moment we first assign a value to it.

B.1.Rules for creating variables in Python are:

1. A variable name must start with a letter or the underscore character.
2. A variable name cannot start with a number.
3. A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
4. Variable names are case-sensitive (name, Name and NAME are three different variables).
5. The reserved words (keywords) cannot be used naming the variable.

Example: *If you give a variable an illegal name, you get a syntax error*

1. >>> 76trombones = 'big parade'
   Output:SyntaxError: invalid syntax
2. >>> more@ = 1000000
   Output: SyntaxError: invalid syntax
3. >>> class = 'Advanced Theoretical Zymurgy'
   Output: SyntaxError: invalid syntax

Ecplaination:76trombones is illegal because it begins with a number.
more@ is illegal because it contains an illegal character, @.

An assignment statement creates new variables and gives them values:This example makes three assignments (*We use the python command (>>>)to start the interpreter. python*)

Example 1(Part A):*A value is one of the basic things a program works with, like a letter or a*

Number)

1. >>>message = 'Hello Python'
2. >>>n = 17
3. >>>pi = 3.1415926535897931

The first assigns a string to a new variable named message.

The second assigns the integer 17 to n.

The third assigns the (approximate) value of π to pi.

Example 1(Part B):*To display the value of a variable, you can use a print statement.*

1. >>> print(n)

Output:17

2. >>> print(pi)

Output:3.141592653589793

Example 1(Part C):*Values and types*(The type of a variable is the type of the value it refers to).

1. >>> type(message)
   Output: <class 'str'>
2. >>> type(n)
   Output: <class 'int'>
3. >>> type(pi)
   Output: <class 'float'>

- Strings belong to the **type str**(You and the interpreter can identify strings because they are enclosed in quotation mark**s**).
- Integers belong to the **type int**.
- Numbers with a decimal point belong to a **type float.**

Example 1(Part D): When you type a large integer, you might be tempted to use commas between

groups of three digits, as in 1,000,000. This is not a legal integer in Python,

but it is legal:

1. >>> print(1,000,000)

Output: 1 0 0 #Well, that's not what we expected at all!

Python interprets 1,000,000 as a comma- separated sequence of integers, which it prints with spaces between.

Note:*This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing***

B.2.Choosing mnemonic variable names

As long as you follow the simple rules of variable naming, and avoid reserved words, you have a lot of choice when you name your variables. In the beginning, this choice can be confusing both when you read a program and when you write your own programs. For example, the following three programs are identical in terms of what they accomplish, but very different when you read them and try to understand them.

Example 1:

```
>>>a = 35.0 b = 12.50 c = a
* b >>>print(c)
```

Example 2:

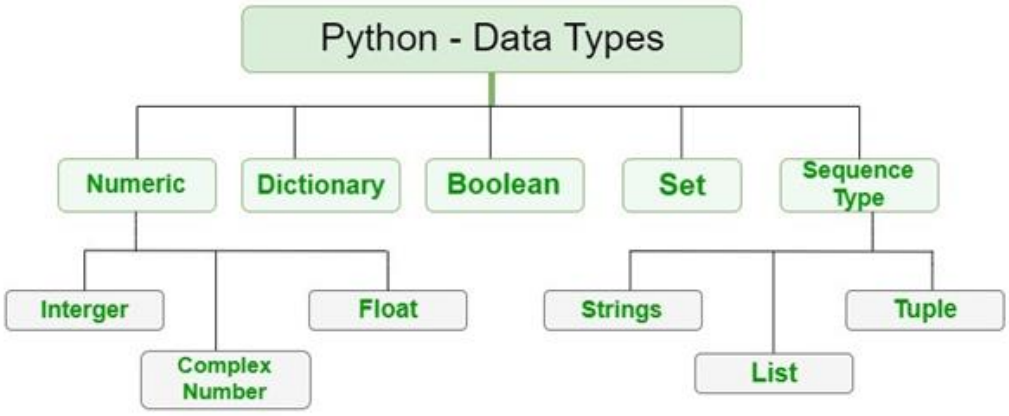
```
>>>hours = 35.0 rate = 12.50
>>>pay = hours * rate
>>>print(pay)
```

The Python interpreter sees all three of these programs as *exactly the same* but humans see and understand these programs quite differently. Humans will most quickly understand the *intent* of the second program because the programmer has chosen variable names that reflect their intent regarding what data will be stored in each variable.

We call these wisely chosen variable names "mnemonic variable names". The word *mnemonic*[1]means "memory aid". We choose mnemonic variable names to help us remember why we created the variable in the first place.

To use mnemonic variable names, mnemonic variable names can get in the way of a beginning programmer's ability to parse and understand code. This is because beginning programmers have not yet memorized the reserved words (there are only 33 of them) and sometimes variables with names that are too descriptive start to look like part of the language and not just well-chosen variable names.

The parts of the code that are defined by Python (for, in, print, and :) are in bold and the programmer-chosen variables (word and words) are not in bold. Many text editors are aware of Python syntax and will color reserved words differently to give you clues to keep your variables and reserved words separate.



At this point, the syntax error you are most likely to make is an illegal variable name, like class and yield, which are keywords, or odd~job and US$, which contain illegal characters.

If you put a space in a variable name, Python thinks it is two operands without an operator:

Example 1:

```
>>>bad name = 5
Output:SyntaxError: invalid syntax
```

Example 2:

```
>>>month= 09
File "<stdin>", line 1 month = 09
         ^
Output: SyntaxError: invalid token
```

Boolean Type – bool

- The bool data type is used to represent boolean values - True or False. The data type can't contain any other value.
- However, Python will again without much issue convert most things to bool.
- bool (9), Python will consider that True, (Anything beyond 1 is treated as True).
- bool(0) will be considered False. (if you assign an empty string, it's treated as False).

```
# Declaring a boolean
>>> some_bool = True

# Printing a boolean's value
>>> some_bool
True

# Checking a boolean's type
>>> type(some_bool)
<class 'bool'>

# Assigning an empty string to a boolean
>>> some_bool = bool('')

# Checking the boolean's value
>>> some_bool
False
```

Complex Numbers – complex

- It is a rarely used data type, and its job is to represent imaginary numbers in a complex pair.

- The character j is used to express the imaginary part of the number, unlike the i more commonly used in math.

Example:

```
#assigning a value
>>> com1=1j
>>> com2 =3+2j
>>>com1+com2
```

Keywords (Python reserves 35 keywords :)

1. Keywords are the reserved words in Python.
2. We cannot use a keyword as a variable name, function name or any other identifier.
3. They are used to define the syntax and structure of the Python language (The interpreter uses keywords to recognize the structure of the program).
4. In Python, keywords are case sensitive***(All the keywords except True, False and None are in lowercase and they must be written as it is).***

and	del	from	None	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	async
def	for	lambda	return	await

Comments (#)

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. ***These notes are called comments,***and in Python they start with the # symbol:

Example:

```
v = 5    #assign 5 to v
```

Statements

A statement is a unit of code that the Python interpreter can execute. We have seen two kinds of statements: print being an expression statement and assignment.

When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one.

A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

Example

```
>>>print(1)
>>>x = 2
>>>print(x)
```

output

1
2

The assignment statement produces no output.

Asking the user for input

Sometimes we would like to take the value for a variable from the user via their keyboard. Python provides a built-in function called **input** that gets input from the keyboard. When this function is called, the program stops and waits for the user to type something. When the user presses Return or Enter, the program resumes and input returns what the user typed as a string.

Example 1:

```
>>>inp= input()
    Some silly stuff #input accepted by user
>>>print(inp)
```

Output: Some silly stuff

Before getting input from the user, it is a good idea to print a prompt telling the user what to input. You can pass a string to input to be displayed to the user before pausing for input:

Example 2:

```
>>>name= input('What is your name?\n')
>>>print(name)
 What is your name?       #input accepted by user
```

Chuck

Output: Chuck

The sequence **\n** at the end of the prompt represents a *newline*, which is a special character that causes a line break. That's why the user's input appears below the prompt.

If you expect the user to type an integer, you can try to convert the return value to int using the int() function:

C.Basic Operators

Python language supports the following types of operators.

1. Arithmetic Operators
2. Comparison (Relational) Operators
3. Assignment Operators
4. Logical Operators
5. Bitwise Operators
6. Membership Operators
7. Identity Operators

1. Python Arithmetic Operators (Assume a=10, b=20)

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after	9//2 = 4 and 9.0//2.0 = 4.0,

	the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) −	-11//3 = -4, -11.0//3 = -4.0

2. Python Comparison Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.(Assume a=10, b=20)

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

3. Python Assignment Operators

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
- =	It subtracts right operand from the left operand and assign	c - = a is

Subtract AND	the result to left operand	equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

4. Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in the binary format their values will be 0011 1100 and 0000 1101 respectively. Following table lists out the bitwise operators supported by Python language with an example each in those, we use the above two variables (a and b) as operands −

a = 0011 1100

b = 0000 1101

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

Shift	the number of bits specified by the right operand.	
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

5. Python Logical Operators

There are following logical operators supported by Python language (Assume a =10 ,b =20)

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

6. Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below

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

7. Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).

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

Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

Sr. No.	Operator & Description
1	**Exponentiation (raise to the power)
2	~ + -Complement, unary plus and minus (method names for the last two are +@ and -@)
3	* / % //Multiply, divide, modulo and floor division
4	+ -Addition and subtraction
5	>><<Right and left bitwise shift
6	&Bitwise 'AND'
7	^ |Bitwise exclusive `OR' and regular `OR'
8	<= <>>=Comparison operators
9	<> == !=Equality operators
10	= %= /= //= -= += *= **=Assignment operators
11	is is notIdentity operators
12	in not inMembership operators
13	not or andLogical operators

D. String data type and string operations

In Python, **Strings** are arrays of bytes representing Unicode characters. However, Python does not have a character data type, a single character .Strings in Python can be created using single quotes or double quotes or even triple quotes.

In Python, individual characters of a String can be accessed by using the method of Indexing. Indexing allows negative address references to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character and so on. Also access characters from front of the string e.g. 0 refers to first character 1 refers to second character.

A	B	D	R	T	N	K	L	P
0	1	2	3	4	5	6	7	8
-9	-8	-7	-6	-5	-4	-3	-2	-1

Example:

```
String1 = "ABCDEFGHI"
print("Initial String: ")
print(String1)

# Printing First character
print("\nFirst character of String is: ")
print(String1[0])

# Printing Last character
print("\nLast character of String is: ")
print(String1[-1])
```

Output:

```
Initial String:
ABCDEFGHI

First character of String is:

A

Last character of String is:

I
```

D.1. string operations

1. string Slicing

To access a range of characters in the String, method of slicing is used. Slicing in a String is done by using a Slicing operator (colon).

Example:

```
String1 = "ABCDEFGHI"
print("Initial String: ")
print(String1)

# Printing 3rd to 8th character
print("\nSlicing characters from 3-8: ")
print(String1[3:8])

# Printing characters between
# 3rd and 2nd last character
print("\nSlicing characters between " + 3rd and 2nd last character: ")
print(String1[3:-2])
```

Output:

```
Initial String:
ABCDEFGHI


First character of String is:
DEFGHI
Last character of String is:
DEFG
```

Note: While accessing an index out of the range will cause an **IndexError**. Only Integers are allowed to be passed as an index, float or other types will cause a **TypeError**.

2. Deleting/Updating from a String

In Python, Updation or deletion of characters from a String is not allowed. This will cause an error because item assignment or item deletion from a String is not supported. Although deletion of entire String is possible with the use of a built-in del keyword. This is because Strings are immutable, hence elements of a String cannot be changed once it has been assigned. Only new strings can be reassigned to the same name.

Example 1: **Updating Entire String**

```
String1 = "Hello, I'm a Python3"
print("Initial String: ")
print(String1)

# Updating a String
String1 = "Welcome to the VIIT"
print("\nUpdated String: ")
print(String1)
```

Output:

```
Initial String:
Hello, I'm a Python3
Updated String:
Welcome to the VIIT
```

Example 2: **Updation of a character:**

```
String1 = "Hello, I'm a Python3"
print("Initial String: ")
print(String1)

# Updating a character
# of the String
String1[2] = 'p'
print("\nUpdating character at 2nd Index: ")
```

```
print(String1)
```

Output: Error

Traceback (most recent call last):
File "/home/360bb1830c83a918fc78aa8979195653.py", line 10, in
String1[2] = 'p'
TypeError: 'str' object does not support item assignment

Example 3: Deleting Entire String

```
String1 ="Hello, I'm a Python3"
print("Initial String: ")
print(String1)

# Deleting a String
# with the use of del
delString1
print("\nDeleting entire String: ")
print(String1)
```

Output: Error

Traceback (most recent call last):

File "/home/e4b8f2170f140da99d2fe57d9d8c6a94.py", line 12, in
print(String1)
NameError: name 'String1' is not defined

Example 4: **Deletion of a character**

```
String1 = "Hello, I'm a Python3"
print("Initial String: ")
print(String1)

# Deleting a character
```

```
# of the String
del String1[2]
print("\nDeleting character at 2nd Index: ")
print(String1)
```

Output:Error

Traceback (most recent call last):
File "/home/499e96a61e19944e7e45b7a6e1276742.py", line 10, in
del String1[2]
TypeError: 'str' object doesn't support item deletion

3. Concatenation

String Concatenation is the technique of **combining two strings**. String Concatenation can be done using many ways.

We can perform string concatenation using following ways:

- Using + operator
- Using join() method
- Using % operator
- Using format() function

Example 1: Using + Operator

It's very easy to use + operator for string concatenation. This operator can be used to add multiple strings together. However, the arguments must be a string.

```
var1 ="Hello "# Defining strings
var2 ="World"
var3 =var1 +var2 # + Operator is used to combine strings
print(var3)
```

Output: Hello World

Here, the variable `var1` stores the string "Hello " and variable `var2` stores the string "World". The + Operator combines the string that is stored in the var1 and var2 and stores in another variable var3.

The +operator works with strings, but it is not addition in the mathematical sense. Instead it performs *concatenation*, which means joining the strings by linking them end to end.

Example 2:

```
first= 10
second= 15
print(first+second)
```

Output:25

Example 3:

```
first= '100'
second= '150'
print(first + second)
```

Output:100150

Example 4: The*****operator also works with strings by multiplying the content of a string by an integer.

```
first= 'Test '
second= 3
print(first * second)
```

Output: Test TestTest

Example 5: **Using join () Method**

```
var1 = "Hello"
var2 = "World"

print("".join([var1, var2])) # join() method is used to combine the strings

var3 = " ".join([var1, var2]) # join() method is used here to combine
                              # the string with a separator Space(" ")

print(var3)
```

Output: HelloWorld

Hello World

Example 6: **Using % Operator**

```
var1 = "Hello"
var2 = "World"
print("% s % s" % (var1, var2))# % Operator is used here to combine the string
```

Output: Hello World

Here, the % Operator combine the string that is stored in the var1 and var2. The %s denotes string data type. The value in both the variable is passed to the string %s and becomes "Hello World".

Example 7: **Using format () function**

str.format () is one of the string formatting methods in Python, which allows multiple substitutions and value formatting. This method lets us concatenate elements within a string through positional formatting.

```
var1 = "Hello"
var2 = "World"
 print("{} {}".format(var1, var2)) # format function is used here to
                                    combine the string
var3 = "{} {}".format(var1, var2) # store the result in another variable
print(var3)
```

Output: HelloWorld
Hello World

NOTE:

For syntax errors, the error messages don't help much.

SyntaxError: invalid syntax and SyntaxError: invalid token, neither of which is very informative.

The runtime error you are most likely to make is a "use before def;" that is, trying to use a variable before you have assigned a value. This can happen if you spell a variable name wrong:

Example:

```
>>>principal= 327.68
>>>interest= principle * rate
Output:NameError: name 'principle' is not defined
```

Variables names are case sensitive, so LaTeXis not the same as latex.

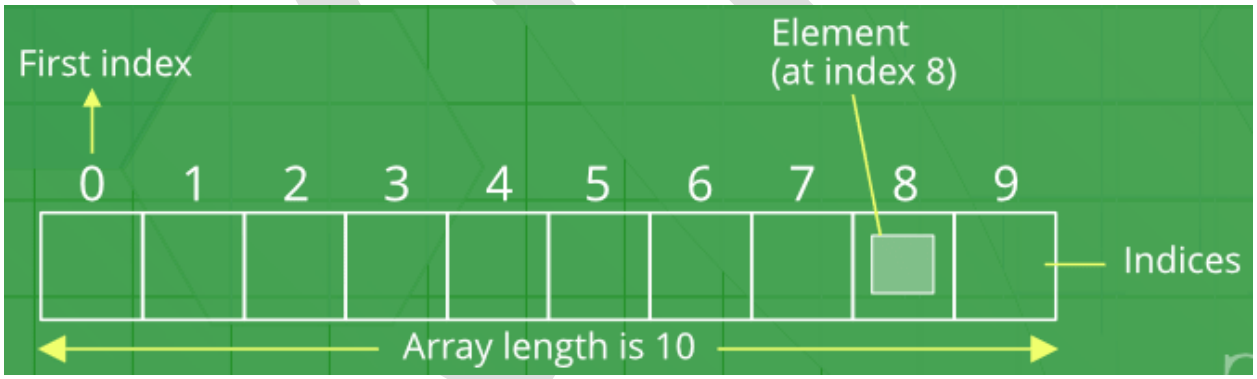
At this point, the most likely cause of a semantic error is the order of operations. For example, to evaluate $1/2\pi$, you might be tempted to write

```
>>>1.0 / 2.0 * pi
```

But the division happens first, so you would get $\pi/2$, which is not the same thing!
There is no way for Python to know what you meant to write, so in this case you don't get an error message; you just get the wrong answer.

E.Python Arrays

- An array is a collection of items stored at contiguous memory locations.
- Array can be handled in Python by a module named **array.**



- Array in Python can be created by importing `array` module. **`Array(data_type, value_list)`** is used to create an array with data type and value list specified in its arguments.

Example

```
import array as arr
```

```
# creating an array with integer type
a = arr.array('i', [1, 2, 3])

# printing original array
print ("The new created array is : ", end =" ")
fori in range (0, 3):
print (a[i], end =" ")
print()

# creating an array with float type
b = arr.array('d', [2.6, 4.1, 7.2])

# printing original array
print ("The new created array is : ", end =" ")
fori in range (0, 3):
print (b[i], end =" ")
```

Output: The new created array is: 1 2 3
The new created array is: 2.6 4.1 7.2

Accessing elements from the Array

In order to access the array items refer to the index number. Use the index operator [ ] to access an item in a array. The index must be an integer.

Example:

```
import array as arr# importing array module

a = arr.array('i', [1, 2, 3, 4, 5, 6]) # array with int type

print("Access element is: ", a[0])# accessing element of array

print("Access element is: ", a[3]) # accessing element of array

b = arr.array('d', [2.5, 3.2, 3.3]) # accessing elemen# array with float type
                                    of array
print("Access element is: ", b[1]) # accessing element of array
print("Access element is: ", b[2])
```

Output: Access element is: 1
Access element is: 4
Access element is: 3.2
Access element is: 3.3

Removing Elements from the Array

remove ():

Elements can be removed from the array by using built-in remove () function but an Error arises if element doesn't exist in the set. `remove ()` method only removes one element at a time, to remove range of elements, iterator is used.

pop()

function can also be used to remove and return an element from the array, but by default it removes only the last element of the array, to remove element from a specific position of the array, index of the element is passed as an argument to the `pop()` method.

Note – Remove method in List will only remove the first occurrence of the searched element.

There are four collection data types in the Python programming language:

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

1.List	2.Tuples	3.Sets	4.Dictionary
List is a collection which is ordered and changeable	**Tuple** is a colection which is ordered and unchangeable.	**Set**is a collection which is unordered and unindexed.	**Dictionary**is a collection which is unordered, changeable and indexed.
Allows duplicate members.	Allows duplicate members.	No duplicate members.	No duplicate members.
written with square brackets []	written with round brackets **( ).**	written with curly brackets **{ }.**	written with curly brackets **{},**and they have keys and values.

Create a List
Example:
 	
L = ["app", "ban", "cher"]
print(L)
	Create a Tuple
	Example:
	
	L = ("app", "ban", "cher")
	print(L)
		Create a Sets
		Example:
		
		L = {"app", "ban", "cher"}
		print(L1)
			Create a Dictionary
			Example
			
			L = {
			"brand": "Ford",
			"model": "Mustang",
			"year": 1964
			}
			print(L)

Output: ['app', 'ban', 'cher']	Output: ('app', 'ban', 'cher')	Output: {'app', 'ban', 'cher'}	**Output:** {'brand': 'Ford', 'model': 'Mustang', 'year': 1964 }
Access the item Example: L = ["app", "ban", "cher"] print(L[2]) Output: cher	**Access the item** Example: L = ("app", "ban", "cher") print(L[1]) Output: ban	**Access the item** Example 1: L = {"app", "ban", "cher"} print(L[0]) Output: TypeError : 'set' object does not support indexing Example 2 L = {"app", "ban", "cher"} for x in L: print(x) Output: app ban cher Example 3 L= {"app", "ban", "cher"} print("ban" in L) Output: True	Access the item Example: x = L["model"] Output: Mustang

List and List slicing

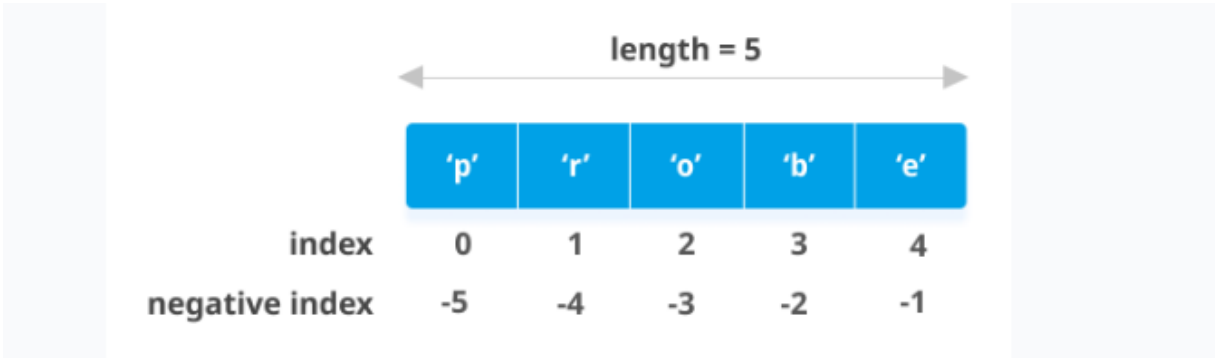
List is one of the most frequently used and very versatile data types used in Python.

We'll learn everything about Python lists, how they are created, slicing of a list, adding or removing elements from them and so on.

How to create a list?

In Python programming, a list is created by placing all the items (elements) inside square brackets [], separated by commas.

It can have any number of items and they may be of different types (integer, float, string etc.).




```
# 1.empty list
My_list= []
# 2.list of integers
My_list = [1, 2, 3]
# 3.list with mixed data types
My_list= [1, "Hello", 3.4]
# 4.nested list
My_list = ["mouse",[ 8, 4, 6],[ ‘a ’]]
```

Access elements from a list

There are various ways in which we can access the elements of a list.

List Index

We can use the index operator [] to access an item in a list. In Python, indices start at 0. So, a list having 5 elements will have an index from 0 to 4.

Trying to access indexes other than these will raise an IndexError. The index must be an integer. We can't use float or other types, this will result in TypeError.

Nested lists are accessed using nested indexing.

Example 1:

```
my_list = ['p', 'r', 'o', 'b', 'e']        # List indexing
print(my_list[0])
print(my_list[2])
print(my_list[4])
```

Output: p
o
e

Example 2:

```
n_list = ["Happy", [2, 0, 1, 5]]     # Nested List
        print(n_list[0][1])
        print(n_list[1][3])
```

Output:a
5

Example 3:

```
n_list = ["Happy", [2, 0, 1, 5]]        # Nested List
print(my_list[4.0])        # Error! Only integer can be used for indexing
```

Output:

```
        Traceback (most recent call last):
         File "<string>", line 10, in <module>
NameError: name 'my_list' is not defined
```

Negative Indexing:

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

Example 1:

```
my_list = ['p','r','o','b','e']   # Negative indexing in lists

print(my_list[-1])

print(my_list[-5])
```

Output:e

p

slice lists in Python

- **We can access a range of items in a list by using the slicing operator** :(colon)**.**
- Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need two indices that will slice that portion from the list.

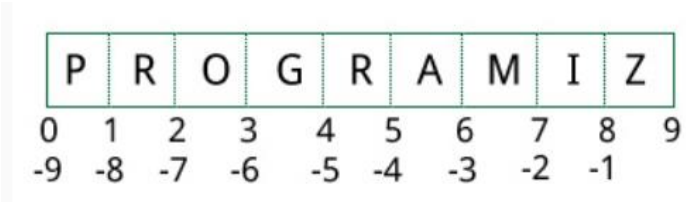


Figure 1Element Slicing from a list in Python

[start: stop: steps]

Which means that slicing will start from index startwill go up to stop in step of steps.
Default value of start is 0, stop is last index of listand for step it is 1

So **[: stop]** will slice list from starting till stop index and **[start :]** will slice list from start index till end Negative value of steps shows right to left traversal instead of left to right traversal that is why **[: : -1]** prints list in reverse order.

Example 1:

```
my_list = ['p','r','o','g','r','a','m','i','z']      # List slicing in Python

        print(my_list[2:5])                    # elements 3rd to 5th

        print(my_list[:-5])                    # elements beginning to 4th

        print(my_list[5:])                     # elements 6th to end

        print(my_list[:])                      # elements beginning to end
```

Output:

```
['o', 'g', 'r']
['p', 'r', 'o', 'g']
['a', 'm', 'i', 'z']
 ['p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z']
```

List manipulation using in build methods

1. Basic List Operations

Lists respond to the **+** and ***** operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1,2,3] : print (x,end = ' ')	1 2 3	Iteration

2. Built-in List Functions and Methods

Python includes the following list functions –

Sr.No.	Function & Description
1	len(list) Gives the total length of the list.
2	max(list) Returns item from the list with max value.
3	min(list) Returns item from the list with min value.
4	list(seq) Converts a tuple into list.

3. Python includes the following list methods

Sr.No.	Methods	Description
1	list.append(obj)	Appends object obj to list
2	list.count(obj)	Returns count of how many times obj occurs in list
3	list.extend(seq)	Appends the contents of seq to list
4	list.index(obj)	Returns the lowest index in list that obj appears
5	list.insert(index, obj)	Inserts object obj into list at offset index

6	list.pop(obj = list[-1])	Removes and returns last object or obj from list
7	list.remove(obj)	Removes object obj from list
8	list.reverse()	Reverses objects of list in place
9	list.sort([func])	Sorts objects of list, use compare func if given

F. Tuple data types

- A tuple is a collection of objects which ordered and immutable.
- Tuples are sequences, just like lists.
- The main difference between the tuples and the lists is that the tuples cannot be changed unlike lists.
- Tuples use parentheses{}, whereas lists use square brackets [].
- Creating a tuple is as simple as putting different comma-separated values. Optionally, you can put these comma-separated values between parentheses also.

Example

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5 )
tup3 = "a", "b", "c", "d"
```

The empty tuple is written as two parentheses containing nothing −

```
tup1 = ();
```

To write a tuple containing a single value you have to include a comma, even though there is only one value −

```
tup1 = (50,)
```

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

Accessing Values in Tuple

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain the value available at that index.

Example

```
tup1 =('physics','chemistry',1997,2000)
tup2 =(1,2,3,4,5,6,7)

print("tup1[0]: ", tup1[0])
print("tup2[1:5]: ", tup2[1:5])
```

Output:tup1[0]: physics
tup2[1:5]: (2, 3, 4, 5)

Updating Tuple

Tuples are immutable, which means you cannot update or change the values of tuple elements. You are able to take portions of the existing tuples to create new tuples.

```
tup1 = (12, 34.56)
tup2 = ('abc', 'xyz')

# Following action is not valid for tuples
# tup1[0] = 100;

# So let's create a new tuple as follows
tup3 = tup1 + tup2
print (tup3)
```

Output:(12, 34.56, 'abc', 'xyz')

Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the **del** statement. For example −

```
tup = ('physics', 'chemistry', 1997, 2000);
print (tup)
del tup;
print ("After deleting tup : ")
```

```
print (tup)
```

Note − An exception is raised. This is because after **del tup**, tuple does not exist anymore.

Output:

```
('physics', 'chemistry', 1997, 2000)
After deleting tup:
Traceback (most recent call last):
 File "test.py", line 9, in <module>
print tup;
NameError: name 'tup' is not defined
```

Basic Tuples Operations

Tuples respond to the **+** **and** ***** operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

Python Expression	**Results**	**Description**
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1,2,3) : print (x, end = ' ')	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Since tuples are sequences, indexing and slicing work the same way for tuples as they do for strings, assuming the following input −

T= **('C++', 'Java', 'Python')**

Python Expression	**Results**	**Description**
T[2]	'Python'	Offsets start at zero

T[-2]	'Java'	Negative: count from the right
T[1:]	('Java', 'Python')	Slicing fetches sections

No Enclosing Delimiters

No enclosing Delimiters is any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples.

Built-in Tuple Functions

Python includes the following tuple functions −

Sr.No.	Function	Description
1	cmp(tuple1, tuple2)	Compares elements of both tuples.
2	len(tuple)	Gives the total length of the tuple.
3	max(tuple)	Returns item from the tuple with max value.
4	min(tuple)	Returns item from the tuple with min value.
5	tuple(seq)	Converts a list into tuple.

G. Dictionary manipulation

- Each key is separated from its value by a colon **(:)**, the items are separated by commas, and the whole thing is enclosed in curly braces**{}.**
- An empty dictionary without any items is written with just two curly braces, like this: {}.
- Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.

Example:

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print ("dict['Name']: ", dict['Name'])
print ("dict['Age']: ", dict['Age'])
```

Output:dict['Name']: Zara
dict['Age']: 7

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry.

Example:

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School" # Add new entry

print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])
```

Output:dict['Age']: 8
dict['School']: DPS School

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

To explicitly remove an entire dictionary, just use the **del** statement.

Example

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

deldict['Name']                    # remove entry with key 'Name'
dict.clear()                       # remove all entries in dict
deldict                        # delete entire dictionary
```

```
print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])
```

Output: An exception is raised because after **deldict**, the dictionary does not exist anymore.

```
dict['Age']:
Traceback (most recent call last):
File "test.py", line 8, in <module>
print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

Note − The del() method is discussed in subsequent section.

Built-in Dictionary Functions and Methods

Sr.No.	Function	Description
1	cmp(dict1, dict2)	No longer available in Python 3.
2	len(dict)	Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
3	str(dict)	Produces a printable string representation of a dictionary
4	type(variable)	Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

Python dictionary methods

Sr.No.	Method	Description
1	dict.clear()	Removes all elements of dictionary *dict*
2	dict.copy()	Returns a shallow copy of dictionary *dict*
3	dict.fromkeys()	Create a new dictionary with keys from seq and values *set* to *value*.
4	dict.get(key, default=None)	For *key* key, returns value or default if key not in dictionary

5	dict.has_key(key)	Removed, use the *in* operation instead.
6	dict.items()	Returns a list of *dict*'s (key, value) tuple pairs
7	dict.keys()	Returns list of dictionary dict's keys
8	dict.setdefault(key, default = None)	Similar to get(), but will set dict[key] = default if *key* is not already in dict
9	dict.update(dict2)	Adds dictionary *dict2*'s key-values pairs to *dict*
10	dict.values()	Returns list of dictionary *dict*'s values

H.Sets:

- A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed).
- A set itself is mutable. We can add or remove items from it.
- A set is created by placing all the items (elements) inside curly braces { }, separated by comma, or by using the built-in set () function.
- It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have mutable elements like lists, sets or dictionaries as its elements.
- Creating an empty set is a bit tricky. Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements, we use the set() function without any argument.

Example 1

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

Output: {'cherry', 'banana', 'apple'}

Note: Sets are unordered, so you cannot be sure in which order the items will appear.

Example 2:

```
my_set = {1, 2, 3}
print(my_set)
```

```
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)
```

Output: {1, 2, 3}
{1.0, (1, 2, 3), 'Hello'}

Example 3:

```
my_set = {1, 2, 3, 4, 3, 2}
print(my_set)
my_set = set([1, 2, 3, 2])
print(my_set)
```

Output: {1, 2, 3, 4}
{1, 2, 3}

Example 4:

```
# distinguish set and dictionary creating empty set
A={ }                    # initialize a with {}
print(type(a))
A=set()                  #initialize a with set()
print(type (A))          # check data type of A
```

Output: <class 'dist'>
<class 'Set'>

Modifying a set in Python

- Sets are mutable. However, since they are unordered, indexing has no meaning.
- We cannot access or change an element of a set using indexing or slicing. Set data type does not support it.
- We can add a single element using the add() method, and multiple elements using the update() method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

```
# initialize my_set
my_set = {1, 3}
print(my_set)
```

```
# if you uncomment line 9,
# you will get an error
# TypeError: 'set' object does not support indexing

# my_set[0]

# add an element
# Output: {1, 2, 3}
my_set.add(2)
print(my_set)

# add multiple elements
# Output: {1, 2, 3, 4}
my_set.update([2, 3, 4])
print(my_set)

# add list and set
# Output: {1, 2, 3, 4, 5, 6, 8}
my_set.update([4, 5], {1, 6, 8})
print(my_set)
```

Output:

```
{1, 3}
{1, 2, 3}
{1, 2, 3, 4}
{1, 2, 3, 4, 5, 6, 8}
```

Removing elements from a set

- A particular item can be removed from a set using the methods discard () and remove ().
- The only difference between the two is that the discard () function leaves a set unchanged if the element is not present in the set. On the other hand, the remove () function will raise an error in such a condition (if element is not present in the set).
- Similarly, we can remove and return an item using the pop() method.
- Since set is an unordered data type, there is no way of determining which item will be popped. It is completely arbitrary.
- We can also remove all the items from a set using the clear() method.

Example 1:

```
# Difference between discard () and remove ()

# initialize my_set
my_set = {1, 3, 4, 5, 6}
print(my_set)

# discard an element
# Output: {1, 3, 5, 6}
my_set.discard(4)
print(my_set)

# remove an element
# Output: {1, 3, 5}
my_set.remove(6)
print(my_set)

# discard an element
# not present in my_set
# Output: {1, 3, 5}
my_set.discard(2)
print(my_set)

# remove an element
# not present in my_set
# you will get an error.
# Output: KeyError

my_set.remove(2)
```

Output

```
{1, 3, 4, 5, 6}
{1, 3, 5, 6}
{1, 3, 5}
{1, 3, 5}
Traceback (most recent call last):
  File "<string>", line 28, in <module>
KeyError: 2
```

Python Set Methods

Method	Description
add()	Adds an element to the set
clear()	Removes all elements from the set
copy()	Returns a copy of the set
difference()	Returns the difference of two or more sets as a new set
difference_update()	Removes all elements of another set from this set
discard()	Removes an element from the set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another
isdisjoint()	Returns `True` if two sets have a null intersection
issubset()	Returns `True` if another set contains this set
issuperset()	Returns `True` if this set contains another set
pop()	Removes and returns an arbitrary set element. Raises `KeyError` if the set is empty
remove()	Removes an element from the set. If the element is not a member, raises a `KeyError`
symmetric_difference()	Returns the symmetric difference of two sets as a new

	set
symmetric_difference_update()	Updates a set with the symmetric difference of itself and another
union()	Returns the union of sets in a new set
update()	Updates the set with the union of itself and others

Built-in Functions with Set

Function	Description
all()	Returns `True` if all elements of the set are true (or if the set is empty).
any()	Returns `True` if any element of the set is true. If the set is empty, returns `False`.
enumerate()	Returns an enumerate object. It contains the index and value for all the items of the set as a pair.
len()	Returns the length (the number of items) in the set.
max()	Returns the largest item in the set.
min()	Returns the smallest item in the set.
sorted()	Returns a new sorted list from elements in the set(does not sort the set itself).
sum()	Returns the sum of all elements in the set.