

Python Mid-Term Outlines Solved

1. Basics

- Overview of Python data types, constants and variables

Constants

Python uses four types of **constants**: integer, floating point, string, and boolean.

1. Integer

Python integers are positive, negative, or zero numbers that do not have a fractional part:

`34 643938 -21 -38472874 0`

Python integers have an unlimited range.

2. Floating Point

Python floating point constants are positive or negative numbers with a fractional part, for example,

`46.1982 0.00441 -237.1273 23.00`

The constant 23.0 is a floating point number, even though it could be converted to an integer without loss of data.

Floating point numbers can also be written in base 10 scientific notation with positive or negative exponents. Here are some examples:

Python Constant	Scientific Notation	Meaning
<code>3.0e2</code>	3.0×10^2	300
<code>5.98e24</code>	5.98×10^{24}	Mass of Earth in Kilos
<code>9.11e-31</code>	9.11×10^{-31}	Mass of Electron in Kilos

The range of float constants is machine dependent, but is usually at least -1.0×10^{300} to 1.0×10^{300} .

Float numbers have at about 15 or 16 significant digits. Such Float numbers are called **double precision** values.

3. String

A Python string is a sequence of characters delimited by single or double quotes.

Examples of string literals:

`"abc" 'g32' "*4^(*$#"`

`" " (space) '' (null string)`

Either single or double quotes can be used for a Python string.

4. Boolean Python datatype `bool`

A boolean constant can take on either of the values `True` or `False`.

Datatypes

- A **datatype** is a Python representation of data.
- Python uses these datatypes to represent constants:

Constant Type	Python Datatype
Integer	<code>int</code>
Floating Point	<code>float</code>
String	<code>str</code>
Boolean	<code>bool</code>

Variables

- A **variable** is a location in a computer that contains data, which might be changed during the execution of a Python script. There are three aspects to a variable: (1) **name**, (2) **value**, (3) **address**.
- The **name** of a Python variable must start with a letter (upper or lowercase) or underscore and consist entirely of letters, digits, or underscores.
- There is no limit to the length of a Python variable name. However, names longer than about 12 characters become unwieldy. Here are some examples of Python variable names:

```
x      x123      customer3      number_of_customers
```

- Python is case sensitive, which means that capitalization matters. The names `number_Of_Customers` and `number_of_customers` represent two different variables.
- By convention, Python spells local variable names in lower case and uses underscores to separate words (**underscore notation**). An example of a local variable name is `number_of_letters`.
- Although a single underscore is a legal variable name, it is not recommended.
- The **address** of a variable is the location in the computer's memory where the data is stored. The Python interpreter keeps a symbol table that maintains the correspondence between the variable's name and its address, so the programmer need not be explicitly concerned about the address.
- The **value** of a variable is the data currently stored at its address. Every variable contains the data of an object. Objects can be from the `int`, `float`, `str`, or `bool` classes. They can also be from any other class defined by the user or from a Python module. A variable can only contain one value at a time.
- The datatype of a variable is inferred by Python by a process called **duck typing**. (If it looks like a duck and quacks like a duck, it's a duck!) For example, consider these Python statements

```
n = 34
x = 3.14
s = "dog"
f = true
```

Python knows that `n` is `int`, `x` is `float`, `s` is `str`, and `f` is `bool` because of the datatypes of the constants on the right hand side of the equation.

- Explicit variable declaration is not used in Python.
- Each variable has a specific type. Although each of the following are allowed:

```
# Following result is 12 because addition is performed
s = 7 + 5

# Following result is "dogcat" because concatenation is performed
t = "dog" + "cat"
```

However, the mixing variable types with the `+` operator is not allowed:

```
s = "328" + 5
```

Use either of the following instead:

```
s = int("328") + 5

t = "dog" + str(5)
```

The Assignment Operator

- The assignment operator is used to assign a value on the right side of an expression to a variable on the left side. For example,

```
x = 5
```

assigns the value 5 to the variable `x`.

- Question: why is the following not a legal assignment?

```
5 = x
```

- Variable naming rules

The Rules

- Variables names must start with a letter or an underscore, such as:
 - `_underscore`
 - `underscore_`
- The remainder of your variable name may consist of letters, numbers and underscores.
 - `password1`
 - `n00b`
 - `underscores`
- Names are case sensitive.
 - `casesensitive`, `CASESENSITIVE`, and `Case_Sensitive` are each a different variable.

The Conventions

- Readability is very important. Which of the following is easiest to read? I'm hoping you'll say the first example.
 - `python_puppet`
 - `pythonpuppet`
 - `pythonPuppet`
- Descriptive names are very useful. If you are writing a program that adds up all of the bad puns made in this book, which do you think is the better variable name?
 - `totalbadpuns`
 - `super_bad`
- Avoid using the lowercase letter 'l', uppercase 'O', and uppercase 'I'. Why? Because the l and the I look a lot like each other and the number 1. And O looks a lot like 0.

- Python reserved words

What are Reserved Keywords in Python?

Python

Server Side Programming

Programming

Reserved words (also called keywords) are defined with predefined meaning and syntax in the language. These keywords have to be used to develop programming instructions. Reserved words can't be used as identifiers for other programming elements like name of variable, function etc.

Following is the list of reserved keywords in Python 3

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

Python 3 has 33 keywords while Python 2 has 30. The print has been removed from Python 2 as keyword and included as built-in function.

To check the keyword list, type following commands in interpreter –

```
>>> import keyword
>>> keyword.kwlist
```

- Numeric expressions and operator precedence

On Lecture Slides

2. Conditional execution

- Comparison operators and Boolean expressions

Boolean Expression in Python

Boolean Expression helps in confirming True OR False given a comparison.

Example

```
> 4 == 4
```

```
True
```

```
> 6 == 2
```

```
False
```

True AND False are not strings. They belong to type bool.

We can validate the type by using the following example code. > type(True)

```
type 'bool'
```

```
> type(False)
```

```
type 'bool'
```

Comparison Operators in Python

Comparison Operators are available for the purpose of comparison. We validate equal to, greater than or less than and so on. To support these kind of comparisons, we use comparison operators. For example, the earlier used '==' is one of the comparison operators.

The following are list of comparison operators:

a == b # it represents a equal to b

a > b # it represents a greater than b

a < b # it represents a less than b

a != b # it represents a not equal to b

a >= b # it represents a greater than or equal to b

a <= b # it represents a less than or equal to b

a is b # it represents a is the same as b

a is not b # it represents a is not the same as b

- Rules for indentation

Indentation in Python

Difficulty Level : Easy • Last Updated : 26 Nov, 2019

Indentation is a very important concept of Python because without proper indenting the Python code, you will end up seeing `IndentationError` and the code will not get compiled.

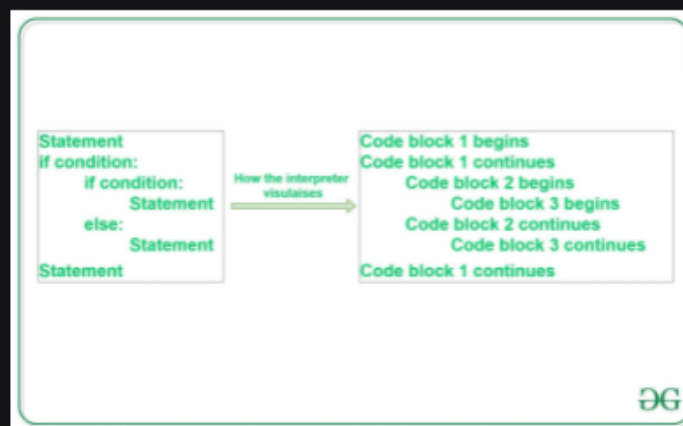
Indentation

In simple terms indentation refers to adding white space before a statement. But the question arises is it even necessary?

To understand this consider a situation where you are reading a book and all of a sudden all the page numbers from the book went missing. So you don't know, where to continue reading and you will get confused. This situation is similar with Python. Without indentation, Python does not know which statement to execute next or which statement belongs to which block. This will lead to `IndentationError`.

Attention geek! Strengthen your foundations with the [Python Programming Foundation](#) Course and learn the basics.

To begin with, your interview preparations Enhance your Data Structures concepts with the [Python DS Course](#). And to begin with your Machine Learning Journey, join the [Machine Learning - Basic Level Course](#)







In the above example,

- Statement (line 1), if condition (line 2), and statement (last line) belongs to the same block which means that after statement 1, if condition will be executed. and suppose the if condition becomes False then the Python will jump to the last statement for execution.
- The nested if-else belongs to block 2 which means that if nested if becomes False, then Python will execute the statements inside the else condition.
- Statements inside nested if-else belongs to block 3 and only one statement will be executed depending on the if-else condition.

Python indentation is a way of telling a Python interpreter that the group of statements belongs to a particular block of code. A block is a combination of all these statements. Block can be regarded as the grouping of statements for a specific purpose. Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation to highlight the blocks of code. Whitespace is used for indentation in Python. All statements with the same distance to the right belong to the same block of code. If a block has to be more deeply nested, it is simply indented further to the right. You can understand it better by looking at the following lines of code.

Example #1:



```
# Python program showing
# indentation

site = 'gfg'

if site == 'gfg':
    print('Logging on to geeksforgeeks...')
else:
    print('retype the URL.')
print('All set !')
```

Output:

```
Logging on to geeksforgeeks...
All set !
```

The lines `print('Logging on to geeksforgeeks...')` and `print('retype the URL.')` are two separate code blocks. The two blocks of code in our example if-statement are both indented four spaces. The final `print('All set!')` is not indented, and so it does not belong to the else-block.

Example #2:

```
j = 1
while(j<= 5):
    print(j)
    j = j + 1
```

Output:

```
1
2
3
4
5
```

To indicate a block of code in Python, you must indent each line of the block by the same whitespace. The two lines of code in the `while` loop are both indented four spaces. It is required for indicating what block of code a statement belongs to. For example, `j=1` and `while(j<=5):` is not indented, and so it is not within `while` block. So, Python code structures by indentation.

Note: Python uses 4 spaces as indentation by default. However, the number of spaces is up to you, but a minimum of 1 space has to be used.

- if, elif and else structures

Python IF...ELIF...ELSE Statements

[⏪ Previous Page](#)

[Next Page ⏩](#)

An **else** statement can be combined with an **if** statement. An **else** statement contains the block of code that executes if the conditional expression in the **if** statement resolves to 0 or a **FALSE** value.

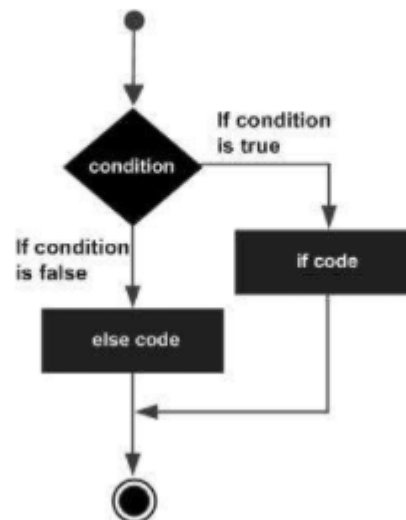
The *else* statement is an optional statement and there could be at most only one **else** statement following **if**.

Syntax

The syntax of the *if...else* statement is –

```
if expression:  
    statement(s)  
else:  
    statement(s)
```

Flow Diagram



Example

```
#!/usr/bin/python

var1 = 100
if var1:
    print "1 - Got a true expression value"
    print var1
else:
    print "1 - Got a false expression value"
    print var1

var2 = 0
if var2:
    print "2 - Got a true expression value"
    print var2
else:
    print "2 - Got a false expression value"
    print var2

print "Good bye!"
```

[Live Demo](#)

When the above code is executed, it produces the following result –

```
1 - Got a true expression value
100
2 - Got a false expression value
0
Good bye!
```

The *elif* Statement

The **elif** statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

Similar to the **else**, the **elif** statement is optional. However, unlike **else**, for which there can be at most one statement, there can be an arbitrary number of **elif** statements following an **if**.

syntax

```
if expression1:
    statement(s)
elif expression2:
    statement(s)
elif expression3:
    statement(s)
else:
    statement(s)
```

Core Python does not provide switch or case statements as in other languages, but we can use `if..elif...` statements to simulate switch case as follows –

Example

```
#!/usr/bin/python

var = 100
if var == 200:
    print "1 - Got a true expression value"
    print var
elif var == 150:
    print "2 - Got a true expression value"
    print var
elif var == 100:
    print "3 - Got a true expression value"
    print var
else:
    print "4 - Got a false expression value"
    print var

print "Good bye!"
```

Live Demo

When the above code is executed, it produces the following result –

```
3 - Got a true expression value
100
Good bye!
```

- Exception handling: try / except structure

On Lecture Slides

3. Working with strings

On Lecture Slides

4. Loops and iterations

- while loop

Python while Loop Statements

[⏪ Previous Page](#)

[Next Page ⏩](#)

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a **while** loop in Python programming language is –

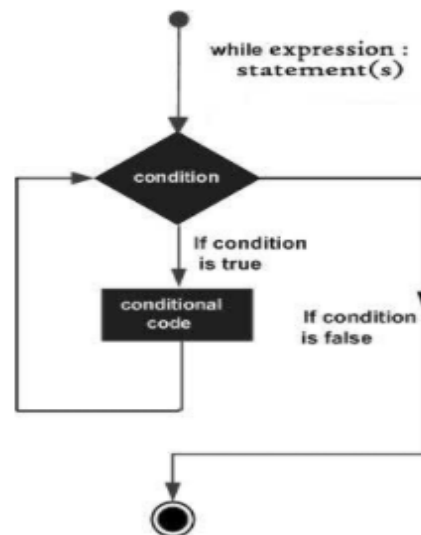
```
while expression:  
    statement(s)
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Flow Diagram



Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

[Live Demo](#)

```
#!/usr/bin/python

count = 0
while (count < 9):
    print 'The count is:', count
    count = count + 1

print "Good bye!"
```

When the above code is executed, it produces the following result –

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!
```

The block here, consisting of the print and increment statements, is executed repeatedly until count is no longer less than 9. With each iteration, the current value of the index count is displayed and then increased by 1.

The Infinite Loop

A loop becomes infinite loop if a condition never becomes FALSE. You must use caution when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.

An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

```
#!/usr/bin/python

var = 1
while var == 1 : # This constructs an infinite loop
    num = raw_input("Enter a number :")
    print "You entered: ", num

print "Good bye!"
```

When the above code is executed, it produces the following result –

When the above code is executed, it produces the following result –

```
Enter a number :20
You entered: 20
Enter a number :29
You entered: 29
Enter a number :3
You entered: 3
Enter a number between :Traceback (most recent call last):
  File "test.py", line 5, in <module>
    num = raw_input("Enter a number :")
KeyboardInterrupt
```

Above example goes in an infinite loop and you need to use CTRL+C to exit the program.

Using else Statement with While Loop

Python supports to have an **else** statement associated with a loop statement.

- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise else statement gets executed.

```
#!/usr/bin/python

count = 0
while count < 5:
    print count, " is less than 5"
    count = count + 1
else:
    print count, " is not less than 5"
```

Live Demo

When the above code is executed, it produces the following result –

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
```

Single Statement Suites

Similar to the **if** statement syntax, if your **while** clause consists only of a single statement, it may be placed on the same line as the while header.

Here is the syntax and example of a **one-line while** clause –

```
#!/usr/bin/python

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

It is better not try above example because it goes into infinite loop and you need to press CTRL+C keys to exit.

- For loop

Python For Loops

[< Previous](#)[Next >](#)

Python For Loops

A **for** loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the **for** keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the **for** loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Example

Print each fruit in a fruit list:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

[Try it Yourself »](#)

The **for** loop does not require an indexing variable to set beforehand.

Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

Example

Loop through the letters in the word "banana":

```
for x in "banana":  
    print(x)
```

[Try it Yourself »](#)

The break Statement

With the `break` statement we can stop the loop before it has looped through all the items:

Example

Exit the loop when `x` is "banana":

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    print(x)  
    if x == "banana":  
        break
```

[Try it Yourself »](#)

Example

Exit the loop when `x` is "banana", but this time the break comes before the print:

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    if x == "banana":  
        break  
    print(x)
```

[Try it Yourself »](#)

The continue Statement

With the `continue` statement we can stop the current iteration of the loop, and continue with the next:

Example

Do not print banana:

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    if x == "banana":  
        continue  
    print(x)
```

[Try it Yourself »](#)

The range() Function

To loop through a set of code a specified number of times, we can use the `range()` function,

The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example

Using the `range()` function:

```
for x in range(6):  
    print(x)
```

[Try it Yourself »](#)

Note that `range(6)` is not the values of 0 to 6, but the values 0 to 5.

The `range()` function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: `range(2, 6)`, which means values from 2 to 6 (but not including 6):

Example

Using the start parameter:

```
for x in range(2, 6):  
    print(x)
```

[Try it Yourself »](#)

The `range()` function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: `range(2, 30, 3)`:

- break and continue statements

Python *break*, *continue* and *pass* Statements

[< Previous Page](#)

[Next Page >](#)

You might face a situation in which you need to exit a loop completely when an external condition is triggered or there may also be a situation when you want to skip a part of the loop and start next execution.

Python provides **break** and **continue** statements to handle such situations and to have good control on your loop.

This tutorial will discuss the *break*, *continue* and *pass* statements available in Python.

The *break* Statement:

The **break** statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C.

The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The **break** statement can be used in both *while* and *for* loops.

Example:

```
#!/usr/bin/python

for letter in 'Python':    # First Example
    if letter == 'h':
        break
    print 'Current Letter :', letter

var = 10                  # Second Example
while var > 0:
    print 'Current variable value :', var
    var = var -1
    if var == 5:
        break

print "Good bye!"
```



This will produce the following result:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

The *continue* Statement:

The **continue** statement in Python returns the control to the beginning of the while loop. The **continue** statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

The **continue** statement can be used in both *while* and *for* loops.

Example:

```
#!/usr/bin/python

for letter in 'Python':    # First Example
    if letter == 'h':
        continue
    print 'Current Letter :', letter

var = 10                  # Second Example
while var > 0:
    var = var -1
    if var == 5:
        continue
    print 'Current variable value :', var
print "Good bye!"
```



This will produce following result:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Good bye!
```

The *else* Statement Used with Loops

Python supports to have an **else** statement associated with a loop statements.

- If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.
- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

Example:

The following example illustrates the combination of an else statement with a for statement that searches for prime numbers from 10 through 20.

```
#!/usr/bin/python

for num in range(10,20): #to iterate between 10 to 20
    for i in range(2,num): #to iterate on the factors of the number
        if num%i == 0:      #to determine the first factor
            j=num/i #to calculate the second factor
            print '%d equals %d * %d' % (num,i,j)
            break #to move to the next number, the #first FOR
    else:                  # else part of the loop
        print num, 'is a prime number'
```



This will produce following result:

```
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
18 equals 2 * 9
19 is a prime number
```

Similar way you can use **else** statement with **while** loop.

The *pass* Statement:

The **pass** statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

The **pass** statement is a *null* operation; nothing happens when it executes. The **pass** is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example):

Example:

```
#!/usr/bin/python

for letter in 'Python':
    if letter == 'h':
        pass
    print 'This is pass block'
    print 'Current Letter :', letter

print "Good bye!"
```



This will produce following result:

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

5. Working with lists

Lists are just like dynamic sized arrays, declared in other languages (vector in C++ and ArrayList in Java). Lists need not be homogeneous always which makes it a most powerful tool in [Python](#). A single list may contain DataTypes like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation.

List in Python are ordered and have a definite count. The elements in a list are indexed according to a definite sequence and the indexing of a list is done with 0 being the first index. Each element in the list has its definite place in the list, which allows duplicating of elements in the list, with each element having its own distinct place and credibility.

Note- Lists are a useful tool for preserving a sequence of data and further iterating over it.

h

Table of content:

- [Creating a List](#)
- [Knowing the size of List](#)
- [Adding Elements to a List:](#)
 - [Using append\(\) method](#)
 - [Using insert\(\) method](#)
 - [Using extend\(\) method](#)
- [Accessing elements from the List](#)
- [Removing Elements from the List:](#)
 - [Using remove\(\) method](#)
 - [Using pop\(\) method](#)
- [Slicing of a List](#)
- [List Comprehension](#)
- [Operations on List](#)
- [List Methods](#)

Creating a List

Lists in Python can be created by just placing the sequence inside the square brackets[]. Unlike [Sets](#), list doesn't need a built-in function for creation of list.

Note – Unlike Sets, list may contain mutable elements.

```
# Python program to demonstrate
# Creation of List

# Creating a List
List = []
print("\nBlank List: ")
print(List)

# Creating a List of numbers
List = [10, 20, 14]
print("\nList of numbers: ")
print(List)

# Creating a List of strings and accessing
# using index
List = ["Geeks", "For", "Geeks"]
print("\nList Items: ")
print(List[0])
print(List[2])

# Creating a Multi-Dimensional List
# (By Nesting a list inside a List)
List = [['Geeks', 'For'], ['Geeks']]
print("\nMulti-Dimensional List: ")
print(List)
```

Output:

Blank List:

```
[]
```

List of numbers:

```
[10, 20, 14]
```

List Items

```
Geeks
```

```
Geeks
```

Multi-Dimensional List:

```
[['Geeks', 'For'], ['Geeks']]
```



Creating a list with multiple distinct or duplicate elements

A list may contain duplicate values with their distinct positions and hence, multiple distinct or duplicate values can be passed as a sequence at the time of list creation.

```
# Creating a List with
# the use of Numbers
# (Having duplicate values)
List = [1, 2, 4, 4, 3, 3, 3, 6, 5]
print("\nList with the use of Numbers: ")
print(List)

# Creating a List with
# mixed type of values
# (Having numbers and strings)
List = [1, 2, 'Geeks', 4, 'For', 6, 'Geeks']
print("\nList with the use of Mixed Values: ")
print(List)
```

Output:

```
List with the use of Numbers:
[1, 2, 4, 4, 3, 3, 3, 6, 5]

List with the use of Mixed Values:
[1, 2, 'Geeks', 4, 'For', 6, 'Geeks']
```

Knowing the size of List

```
# Creating a List
List1 = []
print(len(List1))

# Creating a List of numbers
List2 = [10, 20, 14]
print(len(List2))
```

Output:

```
0
3
```


Adding Elements to a List

Using `append()` method

Elements can be added to the List by using built-in `append()` function. Only one element at a time can be added to the list by using `append()` method, for addition of multiple elements with the `append()` method, loops are used. Tuples can also be added to the List with the use of `append` method because tuples are immutable. Unlike Sets, Lists can also be added to the existing list with the use of `append()` method.

```
# Python program to demonstrate
# Addition of elements in a List

# Creating a List
List = []
print("Initial blank List: ")
print(List)

# Addition of Elements
# in the List
List.append(1)
List.append(2)
List.append(4)
print("\nList after Addition of Three elements: ")
print(List)

# Adding elements to the List
# using Iterator
for i in range(1, 4):
    List.append(i)
print("\nList after Addition of elements from 1-3: ")
print(List)

# Adding Tuples to the List
List.append((5, 6))
print("\nList after Addition of a Tuple: ")
print(List)

# Addition of List to a List
List2 = ['For', 'Geeks']
List.append(List2)
print("\nList after Addition of a List: ")
print(List)
```

Output:

```
Initial blank List:
[]

List after Addition of Three elements:
[1, 2, 4]

List after Addition of elements from 1-3:
[1, 2, 4, 1, 2, 3]

List after Addition of a Tuple:
[1, 2, 4, 1, 2, 3, (5, 6)]

List after Addition of a List:
[1, 2, 4, 1, 2, 3, (5, 6), ['For', 'Geeks']]
```

Using insert() method

append() method only works for addition of elements at the end of the List, for addition of element at the desired position, insert() method is used. Unlike append() which takes only one argument, insert() method requires two arguments(position, value).

```
# Python program to demonstrate
# Addition of elements in a List

# Creating a List
List = [1,2,3,4]
print("Initial List: ")
print(List)

# Addition of Element at
# specific Position
# (using Insert Method)
List.insert(3, 12)
List.insert(0, 'Geeks')
print("\nList after performing Insert Operation: ")
print(List)
```

Output:

```
Initial List:
[1, 2, 3, 4]

List after performing Insert Operation:
['Geeks', 1, 2, 3, 12, 4]
```

Using `extend()` method

Other than `append()` and `insert()` methods, there's one more method for Addition of elements, [`extend\(\)`](#), this method is used to add multiple elements at the same time at the end of the list.

Note – [`append\(\)`](#) and [`extend\(\)`](#) methods can only add elements at the end.

```
# Python program to demonstrate
# Addition of elements in a List

# Creating a List
List = [1,2,3,4]
print("Initial List: ")
print(List)

# Addition of multiple elements
# to the List at the end
# (using Extend Method)
List.extend([8, 'Geeks', 'Always'])
print("\nList after performing Extend Operation: ")
print(List)
```

Output:

```
Initial List:
[1, 2, 3, 4]

List after performing Extend Operation:
[1, 2, 3, 4, 8, 'Geeks', 'Always']
```

Accessing elements from the List

In order to access the list items refer to the index number. Use the index operator [] to access an item in a list. The index must be an integer. Nested list are accessed using nested indexing.

```
# Python program to demonstrate
# accessing of element from list

# Creating a List with
# the use of multiple values
List = ["Geeks", "For", "Geeks"]

# accessing a element from the
# list using index number
print("Accessing a element from the list")
print(List[0])
print(List[2])

# Creating a Multi-Dimensional List
# (By Nesting a list inside a List)
List = [['Geeks', 'For'], ['Geeks']]

# accessing an element from the
# Multi-Dimensional List using
# index number
print("Accessing a element from a Multi-Dimensional list")
print(List[0][1])
print(List[1][0])
```

Output:

```
Accessing a element from the list
Geeks
Geeks
Accessing a element from a Multi-Dimensional list
For
Geeks
```

Negative indexing

In Python, negative sequence indexes represent positions from the end of the array. Instead of having to compute the offset as in `List[len(List)-3]`, it is enough to just write `List[-3]`. Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second-last item, etc.

```

List = [1, 2, 'Geeks', 4, 'For', 6, 'Geeks']

# accessing an element using
# negative indexing
print("Accessing element using negative indexing")

# print the last element of list
print(List[-1])

# print the third last element of list
print(List[-3])

```

Output:

```

Accessing element using negative indexing
Geeks
For

```

Removing Elements from the List

Using `remove()` method

Elements can be removed from the List 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. The `remove()` method removes the specified item.

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

```

# Python program to demonstrate
# Removal of elements in a List

# Creating a List
List = [1, 2, 3, 4, 5, 6,
        7, 8, 9, 10, 11, 12]
print("Initial List: ")
print(List)

# Removing elements from List
# using Remove() method
List.remove(5)
List.remove(6)
print("\nList after Removal of two elements: ")
print(List)

```

```

# Removing elements from List
# using iterator method
for i in range(1, 5):
    List.remove(i)
print("\nList after Removing a range of elements: ")
print(List)

```

Output:

```

Initial List:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

List after Removal of two elements:
[1, 2, 3, 4, 7, 8, 9, 10, 11, 12]

List after Removing a range of elements:
[7, 8, 9, 10, 11, 12]

```

Using pop() method

[Pop\(\)](#) function can also be used to remove and return an element from the set, but by default it removes only the last element of the set, to remove element from a specific position of the List, index of the element is passed as an argument to the pop() method.

```

List = [1,2,3,4,5]

# Removing element from the
# Set using the pop() method
List.pop()
print("\nList after popping an element: ")
print(List)

# Removing element at a
# specific location from the
# Set using the pop() method
List.pop(2)
print("\nList after popping a specific element: ")
print(List)

```

Output:

List after popping an element:

[1, 2, 3, 4]

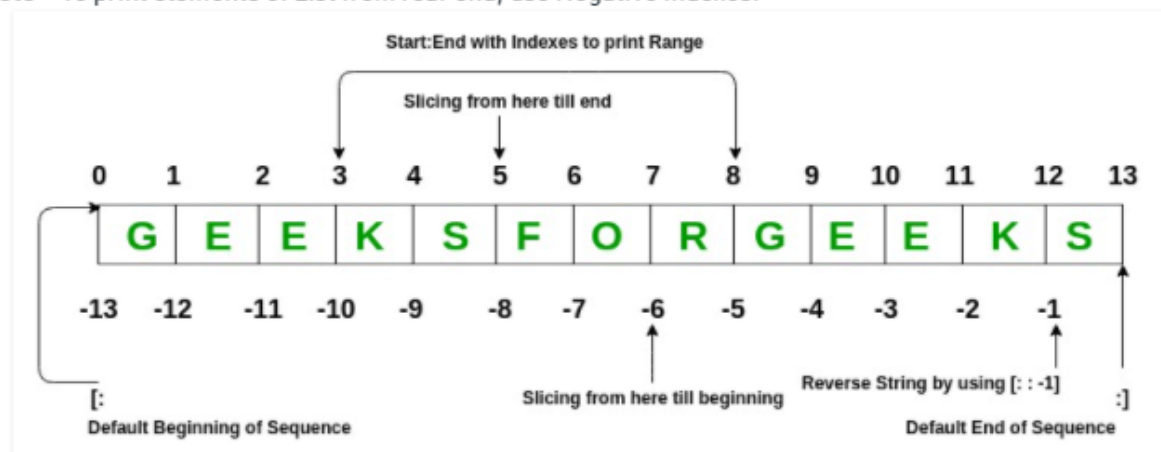
List after popping a specific element:

[1, 2, 4]

Slicing of a List

In Python List, there are multiple ways to print the whole List with all the elements, but to print a specific range of elements from the list, we use [Slice operation](#). Slice operation is performed on Lists with the use of a colon (:). To print elements from beginning to a range use [: Index], to print elements from end-use [: -Index], to print elements from specific Index till the end use [Index:], to print elements within a range, use [Start Index:End Index] and to print the whole List with the use of slicing operation, use [:]. Further, to print the whole List in reverse order, use[::-1].

Note – To print elements of List from rear end, use Negative Indexes.



```
# Python program to demonstrate
# Removal of elements in a List

# Creating a List
List = ['G','E','E','K','S','F','O','R','G','E','E','K','S']

print("Initial List: ")
print(List)

# Print elements of a range
# using Slice operation
Sliced_List = List[3:8]
print("\nSlicing elements in a range 3-8: ")
print(Sliced_List)
```

```

# Print elements from a
# pre-defined point to end
Sliced_List = List[5:]
print("\nElements sliced from 5th "
      "element till the end: ")
print(Sliced_List)

# Printing elements from
# beginning till end
Sliced_List = List[:]
print("\nPrinting all elements using slice operation: ")
print(Sliced_List)

```

Output:

```

Initial List:
['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Slicing elements in a range 3-8:
['K', 'S', 'F', 'O', 'R']

Elements sliced from 5th element till the end:
['F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Printing all elements using slice operation:
['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

```

Negative index List slicing

```

# Creating a List
List = ['G','E','E','K','S','F',
        'O','R','G','E','E','K','S']
print("Initial List: ")
print(List)

# Print elements from beginning
# to a pre-defined point using Slice
Sliced_List = List[:-6]
print("\nElements sliced till 6th element from last: ")
print(Sliced_List)

# Print elements of a range
# using negative index List slicing
Sliced_List = List[-6:-1]
print("\nElements sliced from index -6 to -1")
print(Sliced_List)

```



```

# Printing elements in reverse
# using Slice operation
Sliced_List = List[::-1]
print("\nPrinting List in reverse: ")
print(Sliced_List)

```

Output:

```

Initial List:
['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Elements sliced till 6th element from last:
['G', 'E', 'E', 'K', 'S', 'F', 'O']

Elements sliced from index -6 to -1
['R', 'G', 'E', 'E', 'K']

Printing List in reverse:
['S', 'K', 'E', 'E', 'G', 'R', 'O', 'F', 'S', 'K', 'E', 'E', 'G']

```

List Comprehension

[List comprehensions](#) are used for creating new lists from other iterables like tuples, strings, arrays, lists, etc.

A list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element.

Syntax:

```
newList = [ expression(element) for element in oldList if condition ]
```

Example:

```

# Python program to demonstrate list
# comprehension in Python

# below list contains square of all
# odd numbers from range 1 to 10
odd_square = [x ** 2 for x in range(1, 11) if x % 2 == 1]
print (odd_square)

```



Output:

```
[1, 9, 25, 49, 81]
```

For better understanding the above code is similar to –

```
# for understanding, above generation is same as,  
odd_square = []  
  
for x in range(1, 11):  
    if x % 2 == 1:  
        odd_square.append(x**2)  
  
print (odd_square)
```

Output:

```
[1, 9, 25, 49, 81]
```

List Methods

Function	Description
<u>Append()</u>	Add an element to the end of the list
<u>Extend()</u>	Add all elements of a list to the another list
<u>Insert()</u>	Insert an item at the defined index
<u>Remove()</u>	Removes an item from the list
<u>Pop()</u>	Removes and returns an element at the given index
<u>Clear()</u>	Removes all items from the list
<u>Index()</u>	Returns the index of the first matched item
<u>Count()</u>	Returns the count of number of items passed as an argument
<u>Sort()</u>	Sort items in a list in ascending order
<u>Reverse()</u>	Reverse the order of items in the list
<u>copy()</u>	Returns a copy of the list

Built-in functions with List

Function	Description
<u>reduce()</u>	apply a particular function passed in its argument to all of the list elements stores the intermediate result and only returns the final summation value
<u>sum()</u>	Sums up the numbers in the list
<u>ord()</u>	Returns an integer representing the Unicode code point of the given Unicode character
<u>cmp()</u>	This function returns 1, if first list is "greater" than second list
<u>max()</u>	return maximum element of given list
<u>min()</u>	return minimum element of given list
<u>all()</u>	Returns true if all element are true or if list is empty
<u>any()</u>	return true if any element of the list is true. if list is empty, return false
<u>len()</u>	Returns length of the list or size of the list
<u>enumerate()</u>	Returns enumerate object of list
<u>accumulate()</u>	apply a particular function passed in its argument to all of the list elements returns a list containing the intermediate results
<u>filter()</u>	tests if each element of a list true or not
<u>map()</u>	returns a list of the results after applying the given function to each item of a given iterable
<u>lambda()</u>	This function can have any number of arguments but only one expression, which is evaluated and returned.

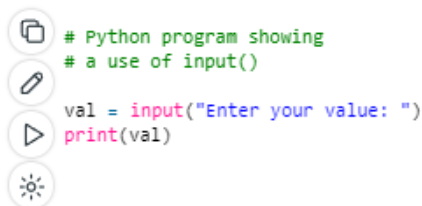
6. Recall and understand the usage of the following built-in functions

- `print()` and `input()`

Developers often have a need to interact with users, either to get data or to provide some sort of result. Most programs today use a dialog box as a way of asking the user to provide some type of input. While Python provides us with two inbuilt functions to read the input from the keyboard.

- `input (prompt)`
- `raw_input (prompt)`

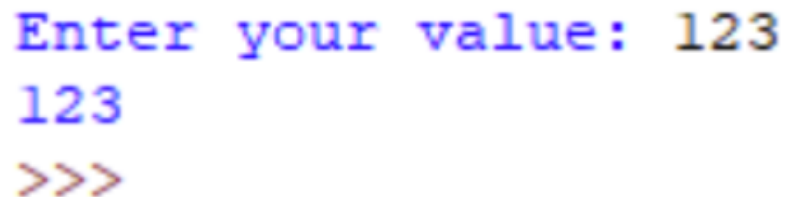
input () : This function first takes the input from the user and then evaluates the expression, which means Python automatically identifies whether user entered a string or a number or list. If the input provided is not correct then either syntax error or exception is raised by python. For example –



```
# Python program showing
# a use of input()

val = input("Enter your value: ")
print(val)
```

Output:



```
Enter your value: 123
123
>>>
```

How the input function works in Python :

- When `input()` function executes program flow will be stopped until the user has given an input.
- The text or message display on the output screen to ask a user to enter input value is optional i.e. the prompt, will be printed on the screen is optional.
- Whatever you enter as input, input function convert it into a string. if you enter an integer value still `input()` function convert it into a string. You need to explicitly convert it into an integer in your code using [typecasting](#).

Code:

```
# Program to check input
# type in Python

num = input ("Enter number :")
print(num)
name1 = input("Enter name : ")
print(name1)

# Printing type of input value
print ("type of number", type(num))
print ("type of name", type(name1))
```

Output :

```
Enter number :123
123
Enter name : geeksforgeeks
geeksforgeeks
type of number <class 'str'>
type of name <class 'str'>
>>> |
```

raw_input () : This function works in older version (like Python 2.x). This function takes exactly what is typed from the keyboard, convert it to string and then return it to the variable in which we want to store. For example –

```
# Python program showing
# a use of raw_input()

g = raw_input("Enter your name : ")
print g
```

Output :

```
Enter your name : geeksforgeeks
geeksforgeeks
>>> |
```

Here, **g** is a variable which will get the string value, typed by user during the execution of program. Typing of data for the `raw_input()` function is terminated by enter key. We can use `raw_input()` to enter numeric data also. In that case we use typecasting. For more details on typecasting refer [this](#).

- `type()` and type-conversion functions, such as `int()`, `float()`, `str()`, etc.

Data Types and Conversion

[<< Previous Note](#) [Next Note >>](#)

On this page: `type()`, `str()`, `int()`, `float()`, `list()`, `tuple()`, and `set()`

Data Types in Python, `tuple`, `set`

Python has many built-in data types. You have seen: `int` (integer), `float` (floating-point number), `str` (string), `list` (list), and `dict` (dictionary). They all have distinct representations:

```
>>> 'hello'      # str
'hello'
>>> 256         # int
256
>>> 1.99        # float
1.99
>>> [1,2,3,4]   # list
[1, 2, 3, 4]
>>> {'a':'apple', 'b':'banana', 'c':'carrot'} # dict
{'a': 'apple', 'c': 'carrot', 'b': 'banana'}
```

Let me introduce here another important data type, which is called **tuple**. A tuple is just like a list except it is fixed (i.e., immutable). It is marked with a pair of parentheses `()`, with each item separated by a comma `,`. In fact, parentheses are not necessary but commas are, as seen in the bottom example.

```
>>> ('gold', 'silver', 'bronze') # tuple
('gold', 'silver', 'bronze')
>>> 1, 2, 3                      # () are optional
(1, 2, 3)
```

Last but not least, **set** is a very important and handy data type. A set is enclosed in curly braces `{ }`, and it is both orderless and duplicate-less.

```
>>> {'Ross', 'Joey', 'Chandeler'} # set in { }
{'Joey', 'Chandeler', 'Ross'}
>>> {'Ross', 'Joey', 'Chandeler', 'Ross', 'Joey'} # Duplicates are ignored
{'Joey', 'Chandeler', 'Ross'}
```

Displaying object type with `type()`

If you are ever unsure of the type of the particular object, you can use the `type()` function:

```
>>> type('hello')
<type 'str'>
>>> type(256)
<type 'int'>
>>> type('256')
<type 'str'>
```

Converting Between Types

Many Python functions are sensitive to the type of data. For example, you cannot concatenate a string with an integer:

```
>>> age = 21
>>> sign = 'You must be ' + age + '-years-old to enter this bar'

Traceback (most recent call last):
  File "<pyshell#71>", line 1, in <module>
    sign = 'You must be ' + age + '-years-old to enter this bar'
TypeError: cannot concatenate 'str' and 'int' objects
```

Therefore, you will often find yourself needing to convert one data type to another. Luckily, conversion functions are easy to remember: the type names double up as a conversion function. Thus, `str()` is the function that converts an integer, a list, etc. to a string, and `list()` is the function that converts something into the list type. For the example above, you would need the `str()` conversion function:

```
>>> age = 21
>>> sign = 'You must be ' + str(age) + '-years-old to enter this bar'
>>> sign
'You must be 21-years-old to enter this bar'
```

Conversion Functions

Below is a table of the conversion functions in Python and their examples.

Function	Converting what to what	Example
<code>int()</code>	string, floating point → integer	<pre>>>> int('2014') 2014 >>> int(3.141592) 3</pre>
<code>float()</code>	string, integer → floating point number	<pre>>>> float('1.99') 1.99 >>> float(5) 5.0</pre>
<code>str()</code>	integer, float, list, tuple, dictionary → string	<pre>>>> str(3.141592) '3.141592' >>> str([1,2,3,4]) '[1, 2, 3, 4]'</pre>
<code>list()</code>	string, tuple, set, dictionary → list	<pre>>>> list('Mary') # list of characters in 'Mary' ['M', 'a', 'r', 'y'] >>> list((1,2,3,4)) # (1,2,3,4) is a tuple [1, 2, 3, 4] >>> list({1, 2, 3}) # {1, 2, 3} is a set [1, 2, 3]</pre>
<code>tuple()</code>	string, list, set → tuple	<pre>>>> tuple('Mary') ('M', 'a', 'r', 'y') >>> tuple([1,2,3,4]) # [] for list, () for tuple (1, 2, 3, 4)</pre>
<code>set()</code>	string, list, tuple → set	<pre>>>> set('alabama') # unique character set from a string {'b', 'm', 'l', 'a'} >>> set([1, 2, 3, 3, 3, 2]) # handy for removing duplicates from list {1, 2, 3}</pre>

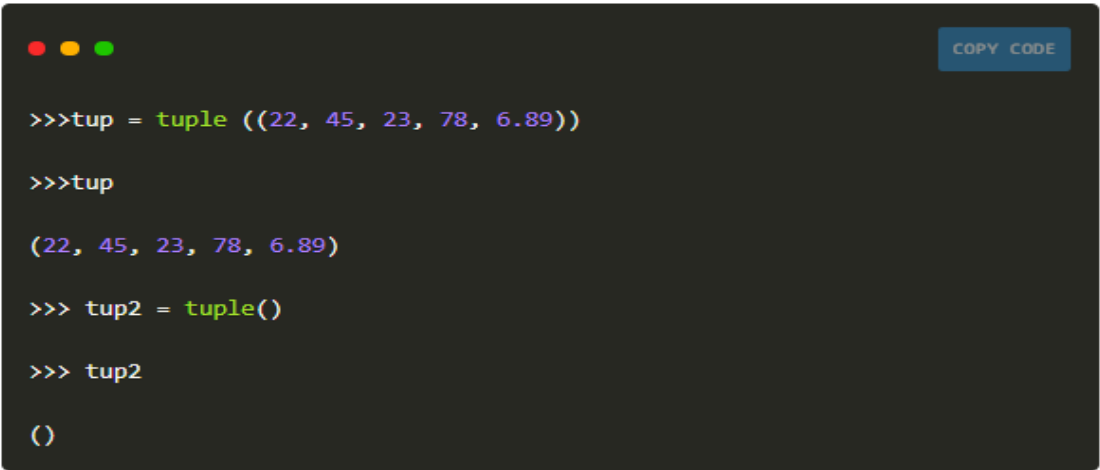
- `len()`, `min()`, `max()`, `sum()`

The tuple() Function

We can use the `tuple()` constructor or function to create a tuple. It basically performs two functions as follows:

- Creating an empty tuple if we give no arguments.
- Creating a tuple with elements if we pass the arguments.

For example,

A screenshot of a Python terminal window with a dark background. It shows the creation of two tuples. The first line is `>>>tup = tuple ((22, 45, 23, 78, 6.89))`, followed by `>>>tup` which returns `(22, 45, 23, 78, 6.89)`. The second line is `>>> tup2 = tuple()`, followed by `>>> tup2` which returns `()`.

```
>>>tup = tuple ((22, 45, 23, 78, 6.89))

>>>tup

(22, 45, 23, 78, 6.89)

>>> tup2 = tuple()

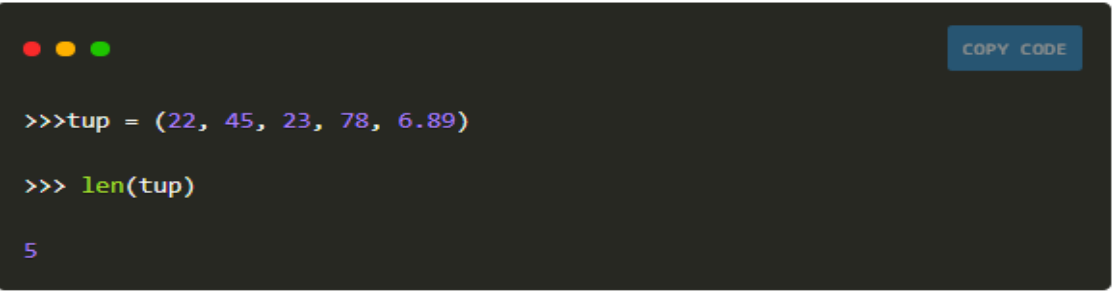
>>> tup2

()
```

The len() Function

This function returns the number of elements present in a tuple. Moreover, it is necessary to provide a tuple to the `len()` function.

For example,

A screenshot of a Python terminal window with a dark background. It shows a tuple being created and then its length being calculated. The first line is `>>>tup = (22, 45, 23, 78, 6.89)`, followed by `>>> len(tup)` which returns `5`.

```
>>>tup = (22, 45, 23, 78, 6.89)

>>> len(tup)

5
```


The count() Function

This function will help us to find the number of times an element is present in the tuple. Furthermore, we have to mention the element whose count we need to find, inside the count function.

For example,

```
>>>tup = (22, 45, 23, 78, 22, 22, 6.89)

>>> tup.count(22)

3

>>> tup.count(54)

0
```

The index() Function

The tuple index() method helps us to find the index or occurrence of an element in a tuple. This function basically performs two functions:

- Giving the first occurrence of an element in the tuple.
- Raising an exception if the element mentioned is not found in the tuple.

For example,

Example 1: Finding the index of an element

```
>>> tup = (22, 3, 45, 4, 2.4, 2, 56, 890, 1)

>>> print(tup.index(45))

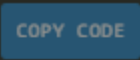

>>> print(tup.index(890))

#prints the index of elements 45 and 890

2

7
```

Example 2:



```
>>> tup = (22, 3, 45, 4, 2.4, 2, 56, 890, 1)

>>> print(tup.index(3.2))

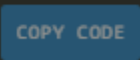

# gives an error because the element is not present in the tuple.

ValueError: tuple.index(x): x not in tuple
```

The sorted() Function

This method takes a tuple as an input and returns a sorted list as an output. Moreover, it does not make any changes to the original tuple.

For example,



```
>>> tup = (22, 3, 45, 4, 2.4, 2, 56, 890, 1)

>>> sorted(tup)

[1, 2, 2.4, 3, 4, 22, 45, 56, 890]
```

The min(), max(), and sum() Tuple Functions

min(): gives the smallest element in the tuple as an output. Hence, the name is min().

For example,

max(): gives the largest element in the tuple as an output. Hence, the name is max().

For example,

[COPY CODE](#)

```
>>> tup = (22, 3, 45, 4, 2.4, 2, 56, 890, 1)
```

```
>>> max(tup)
```

```
890
```

sum(): gives the sum of the elements present in the tuple as an output.

For example,

[COPY CODE](#)

```
>>> tup = (22, 3, 45, 4, 2, 56, 890, 1)
```

```
>>> sum(tup)
```

```
1023
```

- `dir()`

dir() is a powerful inbuilt function in Python3, which returns list of the attributes and methods of any object (say functions, modules, strings, lists, dictionaries etc.)

Syntax:

```
dir({object})
```

Parameters :

`object [optional]` : Takes object name

Returns :

dir() tries to return a valid list of attributes of the object it is called upon. Also, *dir()* function behaves rather differently with different type of objects, as it aims to produce the most relevant one, rather than the complete information.

- For Class Objects, it returns a list of names of all the valid attributes and base attributes as well.
- For Modules/Library objects, it tries to return a list of names of all the attributes, contained in that module.
- If no parameters are passed it returns a list of names in the current local scope.

Code #1 : With and Without importing external libraries.

Python3

```
# Python3 code to demonstrate dir()
# when no parameters are passed

# Note that we have not imported any modules
print(dir())

# Now let's import two modules
import random
import math

# return the module names added to
# the local namespace including all
# the existing ones as before
print(dir())
```

Output :

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
                                '__name__', '__package__', '__spec__']

['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
                                '__name__', '__package__', '__spec__', 'math', 'random']
```

Code #2 :

Python3

```
# Python3 code to demonstrate dir() function
# when a module Object is passed as parameter.

# import the random module
import random

# Prints list which contains names of
# attributes in random function
print("The contents of the random library are::")

# module Object is passed as parameter
print(dir(random))
```

Output :

The contents of the random library are ::

```
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST',
'SystemRandom', 'TWOPI', '_BuiltinMethodType', '_MethodType', '_Sequence',
'_Set', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', '_acos', '_ceil', '_cos', '_e', '_exp',
'_inst', '_log', '_pi', '_random', '_sha512', '_sin', '_sqrt', '_test', '_test_generator',
'_urandom', '_warn', 'betavariate', 'choice', 'expovariate', 'gammavariate', 'gauss',
'getrandbits', 'getstate', 'lognormvariate', 'normalvariate', 'paretovariate', 'randint',
'random', 'randrange', 'sample', 'seed', 'setstate', 'shuffle', 'triangular', 'uniform',
'vonmisesvariate', 'weibullvariate']
```

Code #3 : Object is passed as parameters.

Python3

```
# When a list object is passed as
# parameters for the dir() function

# A list, which contains
# a few random values
geeks = ["geeksforgeeks", "gfg", "Computer Science",
        "Data Structures", "Algorithms" ]

# dir() will also list out common
# attributes of the dictionary
d = {} # empty dictionary

# dir() will return all the available
# list methods in current local scope
print(dir(geeks))

# Call dir() with the dictionary
# name "d" as parameter. Return all
# the available dict methods in the
# current local scope
print(dir(d))
```

Output :

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
 '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__',
 '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
 '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setitem__',
 '__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'items',
 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

Code #4 : User Defined – Class Object with an available `__dir()` method is passed as parameter.

Python3

```
# Python3 program to demonstrate working
# of dir(), when user defined objects are
# passed are parameters.

# Creation of a simple class with __dir__()
# method to demonstrate it's working
class Supermarket:

    # Function __dir__() which list all
    # the base attributes to be used.
    def __dir__(self):
        return ['customer_name', 'product',
                'quantity', 'price', 'date']

# user-defined object of class supermarket
my_cart = Supermarket()

# listing out the dir() method
print(dir(my_cart))
```

Output :

```
['customer_name', 'date', 'price', 'product', 'quantity']
```

Applications :

- The **dir()** has it's own set of uses. It is usually used for *debugging purposes* in simple day to day programs, and even in large projects taken up by a team of developers. The capability of `dir()` to list out all the attributes of the parameter passed, is really useful when handling a lot of classes and functions, separately.
- The **dir()** function can also list out all the available attributes for a module/list/dictionary. So, it also gives us information on the operations we can perform with the available list or module, which can be very useful when having little to no information about the module. It also helps to know new modules faster.