Python Notes

Unit - 1

Python is powerful... and fast; plays well with others; runs everywhere; is friendly & easy to learn; is Open.

These are some of the reasons people who use Python would rather not use anything else.

Getting Started

Python can be easy to pick up whether you're a first time programmer or you're experienced with other languages. The following pages are a useful first step to get on your way writing programs with Python!

Beginner's Guide, Programmers

Beginner's Guide, Non-Programmers

Beginner's Guide, Download & Installation

Code sample and snippets for Beginners

Friendly & Easy to Learn

The community hosts conferences and meetups, collaborates on code, and much more. Python's documentation will help you along the way, and the mailing lists will keep you in touch.

Conferences and Workshops

Python Documentation

Mailing Lists and IRC channels

Applications

The Python Package Index (PyPI) hosts thousands of third-party modules for Python. Both Python's standard library and the community-contributed modules allow for endless possibilities.

Web and Internet Development

Database Access

Desktop GUIs

Scientific & Numeric

Education

Network Programming

Software & Game Development

Open-source

Python is developed under an OSI-approved open source license, making it freely usable and distributable, even for commercial use. Python's license is administered by the Python Software Foundation.

Learn more about the license

Python license on OSI

Learn more about the Foundation

Getting Started

Python can be easy to pick up whether you're a first time programmer or you're experienced with other languages. The following pages are a useful first step to get on your way writing programs with Python!

Beginner's Guide, Programmers

Beginner's Guide, Non-Programmers

Beginner's Guide, Download & Installation

Code sample and snippets for Beginners

Friendly & Easy to Learn

The community hosts conferences and meetups, collaborates on code, and much more. Python's documentation will help you along the way, and the mailing lists will keep you in touch.

Conferences and Workshops

Python Documentation

Mailing Lists and IRC channels

Applications

The Python Package Index (PyPI) hosts thousands of third-party modules for Python. Both Python's standard library and the community-contributed modules allow for endless possibilities.

Web and Internet Development

Database Access

Desktop GUIs

Scientific & Numeric

Education

Network Programming

Software & Game Development

Open-source

Python is developed under an OSI-approved open source license, making it freely usable and distributable, even for commercial use. Python's license is administered by the Python Software Foundation.

Learn more about the license

Python license on OSI

Learn more about the Foundation

Mathematical Operators

```
>>> 2+3                    (Addition)
5

>>> 5-4                    (Subtraction)
1

>>> 2*9                    (Multiplication)
18

>>> 4/2                    (Division)  -  Division operator by default returns Float Value (Decimal
Value )
2.0

>>> 5/2                    (Division)
2.5

>>> 5//2                   (Floor Division) – Provides Floor Value of division
2

>>> 2**3                   (Power – of )  2^3
8

>>> 10%3                   (Modulus) – Return the reminder of the division
1
```

```
>>> x=5
>>> y=7
>>> x
5
>>> _
5
>>> x + _
10
 >>> _
10
```

Underscore (_) used as a hidden variable in python which simply stores the last returned value. We can also assign it as a variable.

print Function

```
>>> 'python'
'python'

>>> "python"
'python'

>>> '25'
'25'                                    -   (i)

>>> 25
25                                      -   (ii)
```

Can you differentiate between (i) and (ii), actually by seeing them we can know thier data types. It is clearly understandable that (i) is of string data type as it is in apostrophe (‘ ‘) where as (ii) is of integer data type.

The **print function** in **Python** is a **function** that outputs to your console window whatever you say you want to **print** out. At first blush, it might appear that the **print function** is rather useless for programming, but it is actually one of the most widely **used functions** in all of **python**.

Print function print/display as it is value that is passed in it irrespective of it’s data type.

```
>>> print('hello python')
hello python

>>> print('23')                          (iii)
23

>>> print(23)                    (iv)
23
```

It is clear that value passed in (iii) is of string type and the value passed in (iv) is of integer type, but it is there is no difference in thier printed value.

```
>>> x = 5
>>> y = 8
>>> print(x+y)
13

>>> print('Python’s Student')
```

SyntaxError: invalid syntax

Here the error is, when the first apostrophe starts compilers read it as the start of the string and when it finds second apostrophe compiler consider it as the end of string and hence the characters after that is not considered in string and gives error.
There are two ways to solve this problem.

1: As we know we can use both (‘ ‘) as well as (“ “) to enter a string and hence we can use (“ “) instead of (‘’).

```
>>> print("Python's Student")
```

2: Second method is a standard approach provided in Python that we can use black slash (\ ) before apostrophe for directing the compiler to ignore that special meaning of apostrophe and print that as a normal string character.

```
Python’s Student
>>> print('Python\'s Student')

Python’s Student
```

String

String literals in **python** are surrounded by either single quotation marks, or double quotation marks.**Strings** can be output to screen using the print function. For example: print("hello"). ... However,**Python** does not have a character data type, a single character is simply a **string** with a length of 1.

```
>>> x = 'Techsim+'
```

We can add two Strings

```
>>> x + ' : The Symbol of Expertise.'
'Techsim+ : The Symbol of Expertise.'
```

```
>>> x*5
'HelloHelloHelloHelloHello
```

#----- Indexing in Python

```
>>> x = 'Hello Python'
>>> x[1]
'e'
>>> x[0]
'H'
>>> x[6]
'P'
>>> x[-1]
'n'
>>> x[-3]
'h'
```

0	1	2	3	4	5	6	7	8	9	10	11
H	**e**	**l**	**l**	**o**		**P**	**y**	**t**	**h**	**o**	**n**
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

In Python we can assign position or we can say indexing from both the sides.

Slicing in String

Slicing creates a **slice** of an object representing the set of indices specified by range (start : stop : step).

```
x = 'Hello Python'
>>> x[4:9]
'o Pyt'
>>> x[6:]
'Python'
>>> x[:9]
'Hello Pyt'

>>> y = [1,2,3,4,5,6,7,8,9]
>>> y[3:8]
[4, 5, 6, 7, 8]3
>>> y[4:-2]
[5, 6, 7]
>>> y[0:8:2]
[1, 3, 5, 7]
>>> y[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Example Program:

Source code :

```
inp = input('enter an URL ')
dom = inp[12:-4]
print('Domain name is :',dom)
```

Output

```
enter an URL https://www.Techsimplus.com
Domain name is : Techsimplus
```

Split function in String

split method returns a list of strings after breaking the given string by the specified separator.

```
>>> x.split()
['Hello', 'Python']

>>> x.split('o')
['Hell', ' Pyth', 'n']
```

By default split function splits from space ( ), but we can also pass a parameter if we want to split the string from any particular character.

String → List And Join function

list() function is used to convert a string into list taking individual character (including space) as elements.

For Example:

```
>>> x = 'Hello Python'
>>> list(x)
['H', 'e', 'l', 'l', 'o', ' ', 'P', 'y', 't', 'h', 'o', 'n']
```

join() function is used to join elements of a list to create a whole string.
We can also pass space, characters, symbols to put in between the elements.

```
>>> y = 'Techsim+'
>>> lis = list(y)
>>> lis
['T', 'e', 'c', 'h', 's', 'i', 'm', '+']
>>> ''.join(lis)
'Techsim+'

>>>x = ‘Hello Python’
>>> lis1 = list(x)
>>> lis1
['H', 'e', 'l', 'l', 'o', ' ', 'P', 'y', 't', 'h', 'o', 'n']
>>> ' '.join(lis1)
'H e l l o   P y t h o n'
>>> '*'.join(lis1)
'H*e*l*l*o* *P*y*t*h*o*n'
```

Importing Libraries in Python

Python modules can get access to code from another **module** by **importing** the file/function using **import**. The **import** statement is the most common way of invoking the **import** machinery.

```
>>> import math
>>> math.sqrt(81)
9.0
>>> math.pow(3,2)
9.0
>>> math.pow(5,4)
625.0
```

Instead of import whole library we can also import only some function so that we don't need to write library name before using any of its function.

```
>>> from math import sqrt, pow
>>> pow(2,5)
32.0
>>> sqrt(36)
6.0
```

If we want to use so many functions of any library but doesn't want to mention library name every time then we can import all of its functions.

```
>>> from math import *
>>> ceil(35.1)
36
>>> floor(75.87)
75
>>> sin(72)
0.25382336276203626
```

```
>>> from math import sqrt as under_root
>>> under_root(9)
3.0
```

List in Python

List is a collection which is ordered and changeable. Allows duplicate members. We can make a list simply by passing element in square brackets [], and it is mutable.

```
>>> x = [1,3,56,78,3,69]

>>> y = ['python', 'django', 'pandas', 'turtle', 'anaconda']

>>> z = [1,'hello',54.35,34,85.3,'web']
```

We can find elements of list through its position/index same as in case of string.

```
>>> x[5]
69

>>> x[-2]
3

>>> y[1]
'django'
```

Play with List using its Functions

```
>>> x = [1,3,56,78,3,69]
```

- Append function is used to insert an element at the end of the list.

```
>>> x.append(45)
>>> x
[1, 3, 56, 78, 3, 69, 45]
```

- Insert function is used to insert an element at a particular place/position.
  for this we need to pass position/indexing first before entering the data to specify where would be the data/element would be added.

```
>>> x.insert(2,77)
>>> x
[1, 3, 77, 56, 78, 3, 69, 45]
```

- The `extend()` method adds the specified list elements (or any iterable) to the end of the current list.

```
>>> li = [1,2,3,4,5,6,7]
>>> li1 = [8,9,10]
>>> li.extend(li1)
>>> li
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Remove function is used to remove an element from a list on the basis of value i.e., we need to enter the element that we need to remove instead of its position.

```
>>> x.remove(78)
>>> x
[1, 3, 77, 56, 3, 69, 45]
```

- Pop function is used to delete an element from a list on the basis of position. By default if we didn't pass any position in it, this function will automatically remove the last element of list.

```
>>> x.pop()
45
>>> x.pop(3)
56
```

- Count function is used to count the reoccurrence of any element in a list.

```
>>> x.count(3)
2
```

- Sort function is used to sort an list, by default it sort's a list in ascending order but we can also sort it in descending order using method given below.

```
>>> x.sort()
>>> x
```

```
[1, 3, 3, 69, 77]
>>> x.sort(reverse = True)
>>> x
[77, 69, 3, 3, 1]
```

- Reverse function is used to reverse an list.

```
>>> x = [1,3,56,78,3,69]
>>> x.reverse()
>>> x
[69, 3, 78, 56, 3, 1]
```

- In simple terms, index() method finds the given element in a list and returns its position.
  However, if the same element is present more than once, index() method returns its smallest/first position.

```
>>> li = ['a','e','i','o','u']
>>> li.index('i')
2
```

- del function is used to delete elements by position. We can also delete multiple elements by passing range in del function.

```
>>> del x[2]
>>> x
[69, 3, 56, 3, 1]

x = [69, 3, 78, 56, 3, 1]
>>> del x[2:6]
>>> x
[69, 3]
```

- min function is used to find minimum valued element in list.

```
>>> x
[69, 3, 56, 3, 1]
>>> min(x)
```

1

- max function is used to find maximum valued element in list.

```
>>> max(x)
69
```

- len function is used to find the length of an list. It returns the total number of elements present in list.

```
>>> len(x)
5
```

Unit - 2

Conditional Statements

Python supports the usual logical conditions from mathematics:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written by using the `if` keyword.

Example

If statement:

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

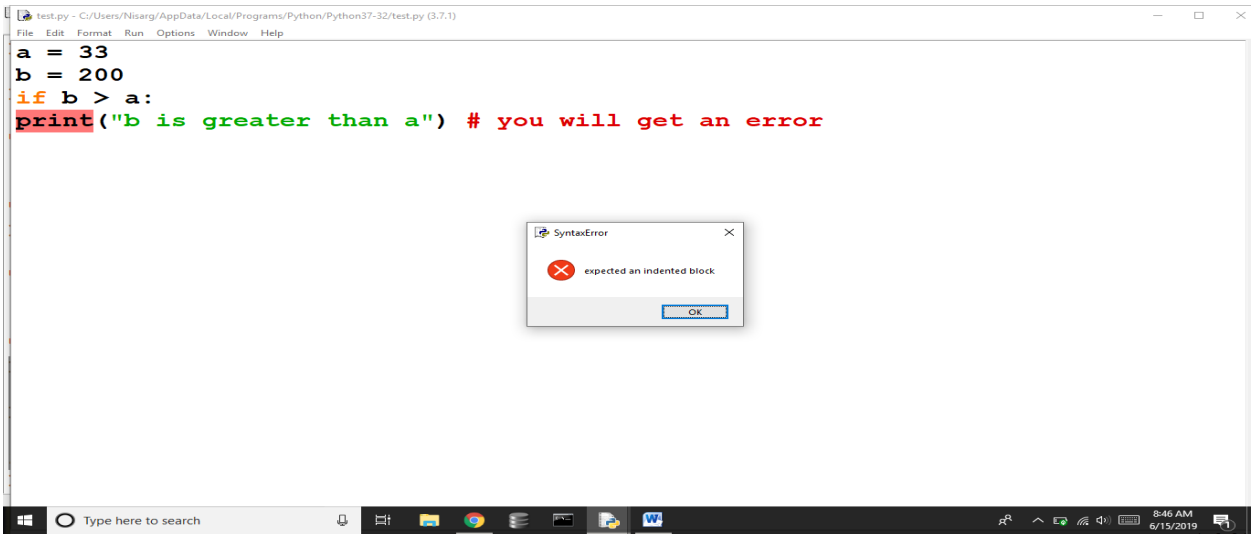
In this example we use two variables, `a` and `b`, which are used as part of the if statement to test whether `b` is greater than `a`. As `a` is `33`, and `b` is `200`, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

Indentation

Python relies on indentation, using whitespace, to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Example

If statement, without indentation (will raise an error):



Elif

The `elif` keyword is pythons way of saying "if the previous conditions were not true, then try this condition".

Example

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

In this example `a` is equal to `b`, so the first condition is not true, but the `elif` condition is true, so we print to screen that "a and b are equal".

Else

The `else` keyword catches anything which isn't caught by the preceding conditions.

Example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

In this example `a` is greater to `b`, so the first condition is not true, also the `elif` condition is not true, so we go to the `else` condition and print to screen that "a is greater than b".

You can also have an `else` without the `elif`:

Example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

Example

One line if statement:

```
if a > b: print("a is greater than b")
```

Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Example

One line if else statement:

```
print("A") if a > b else print("B")
```

You can also have multiple else statements on the same line:

Example

One line if else statement, with 3 conditions:

```
print("A") if a > b else print("=") if a == b else print("B")
```

And

The `and` keyword is a logical operator, and is used to combine conditional statements:

Example

Test if `a` is greater than `b`, AND if `c` is greater than `a`:

```
if a > b and c > a:
  print("Both conditions are True")
```

Or

The `or` keyword is a logical operator, and is used to combine conditional statements:

Example

Test if `a` is greater than `b`, OR if `a` is greater than `c`:

```
if a > b or a > c:
  print("At least one of the conditions is True")
```

LOOPs in Python

For loop

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.

```
>>> for i in Name:
                print(i)

Sourabh
Mahesh
Pranay
Arpit
Lovelesh
Anirudh
```

Looping Through a String

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

Example

Loop through the letters in the word

```
>>> for i in 'Hello Programmers':
                print(i)


H
e
l
l
o

P
r
o
g
r
a
m
m
e
r
s
```

- By default in loop, print function ends with 'end = \n' parameter which means to go to next line but if we want to print in same line we can change that to 'end = '' '

```
>>> for i in 'Hello Programmers':
                print(i, end='')

Hello Programmers
```

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
```

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)
```

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)
```

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)
```

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)
```

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)`

Example

Increment the sequence with 3 (default is 1):

```
for x in range(2, 30, 3):
  print(x)
```

Else in For Loop

The `else` keyword in a `for` loop specifies a block of code to be executed when the loop is finished:

Example

Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

Print each adjective for every fruit:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```

The while Loop

With the `while` loop we can execute a set of statements as long as a condition is true.

Example

Print i as long as i is less than 6:

```
i = 1
while i < 6:
  print(i)
  i += 1
```

Note: remember to increment i, or else the loop will continue forever.

The `while` loop requires relevant variables to be ready, in this example we need to define an indexing variable, `i`, which we set to 1.

The break Statement

With the `break` statement we can stop the loop even if the while condition is true:

Example

Exit the loop when i is 3:

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

The continue Statement

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

Example

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

Unit - 3

Set

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

Example

Create a Set:

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

Note: Sets are unordered, so the items will appear in a random order.

Access Items

You cannot access items in a set by referring to an index, since sets are unordered the items has no index.

But you can loop through the set items using a `for` loop, or ask if a specified value is present in a set, by using the`in` keyword.

Example

Loop through the set, and print the values:

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

Example

Check if "banana" is present in the set:

```
thisset = {"apple", "banana", "cherry"}

print("banana" in thisset)
```

Change Items

Once a set is created, you cannot change its items, but you can add new items.

Add Items

To add one item to a set use the `add()` method.

To add more than one item to a set use the `update()` method.

Example

Add an item to a set, using the `add()` method:

```
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)
```

Example

Add multiple items to a set, using the `update()` method:

```
thisset = {"apple", "banana", "cherry"}

thisset.update(["orange", "mango", "grapes"])

print(thisset)
```

Get the Length of a Set

To determine how many items a set has, use the `len()` method.

Example

Get the number of items in a set:

```
thisset = {"apple", "banana", "cherry"}

print(len(thisset))
```

Remove Item

To remove an item in a set, use the `remove()`, or the `discard()` method.

Example

Remove "banana" by using the `remove()` method:

```
thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")

print(thisset)
```

Note: If the item to remove does not exist, `remove()` will raise an error.

Example

Remove "banana" by using the `discard()` method:

```
thisset = {"apple", "banana", "cherry"}

thisset.discard("banana")

print(thisset)
```

Note: If the item to remove does not exist, `discard()` will **NOT** raise an error.

You can also use the `pop()`, method to remove an item, but this method will remove the *last* item. Remember that sets are unordered, so you will not know what item that gets removed.

The return value of the `pop()` method is the removed item.

Example

Remove the last item by using the `pop()` method:

```
thisset = {"apple", "banana", "cherry"}

x = thisset.pop()

print(x)

print(thisset)
```

Note: Sets are *unordered*, so when using the `pop()` method, you will not know which item that gets removed.

Example

The `clear()` method empties the set:

```
thisset = {"apple", "banana", "cherry"}

thisset.clear()

print(thisset)
```

Example

The `del` keyword will delete the set completely:

```
thisset = {"apple", "banana", "cherry"}

del thisset

print(thisset)
```

Tuple

A tuple is a collection which is ordered and **unchangeable**. In Python tuples are written with round brackets.

Example

Create a Tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

Access Tuple Items

You can access tuple items by referring to the index number, inside square brackets:

Example

Return the item in position 1:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

Change Tuple Values

Once a tuple is created, you cannot change its values. Tuples are **unchangeable**.

Loop Through a Tuple

You can loop through the tuple items by using a `for` loop.

Example

Iterate through the items and print the values:

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

You will learn more about `for` loops in out Python For Loops Chapter.

Check if Item Exists

To determine if a specified item is present in a tuple use the `in` keyword:

Example

Check if "apple" is present in the tuple:

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

Tuple Length

To determine how many items a tuple has, use the `len()` method:

Example

Print the number of items in the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

Add Items

Once a tuple is created, you cannot add items to it. Tuples are **unchangeable**.

Example

You cannot add items to a tuple:

```
thistuple = ("apple", "banana", "cherry")
thistuple[3] = "orange" # This will raise an error
print(thistuple)
```

Remove Items

Note: You cannot remove items in a tuple.

Tuples are **unchangeable**, so you cannot remove items from it, but you can delete the tuple completely:

Example

The `del` keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

Unit - 4

Functions in Python

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Creating a Function

In Python a function is defined using the `def` keyword:

Example

```
def my_function():
  print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

Example

```
def my_function():
  print("Hello from a function")

my_function()
```

Parameters

Information can be passed to functions as parameter.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a function with one parameter (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without parameter, it uses the default value:

Example

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Passing a List as a Parameter

You can send any data types of parameter to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as a parameter, it will still be a List when it reaches the function:

Example

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

Return Values

To let a function return a value, use the `return` statement:

Example

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Types of function:

i) No Input No Output
ii) No Input Yes Output
iii) Yes Input No Output
iv) Yes Input Yes Output

```
##########   No Input No Output   ##########

def Greeting():
    print('Hello')


##########   No Input Yes Output  ##########

def Greeting():
    return 'Hello'


##########  Yes Input No Output   ##########

def Greeting(x):
    x = input('Enter')
    print(x)


##########  Yes Input Yes Output  ##########

def Greeting(x):
    x = input('Enter')
    return x
```

Python Classes and Objects

Python Classes/Objects

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword `class`:

Example

Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
```

Create Object

Now we can use the class named myClass to create objects:

Example

Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

The __init__() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in __init__() function.

All classes have a function called __init__(), which is always executed when the class is being initiated.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Example

Create a class named Person, use the __init__() function to assign values for name and age:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Let us create a method in the Person class:

Example

Insert a function that prints a greeting, and execute it on the p1 object:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Note: The `self` parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

The self Parameter

The `self` parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named `self` , you can call it whatever you like, but it has to be the first parameter of any function in the class:

Example

Use the words *mysillyobject* and *abc* instead of *self*:

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

Modify Object Properties

You can modify properties on objects like this:

Example

Set the age of p1 to 40:

```
p1.age = 40
```

Delete Object Properties

You can delete properties on objects by using the `del` keyword:

Example

Delete the age property from the p1 object:

```
del p1.age
```

Delete Objects

You can delete objects by using the `del` keyword:

Example

Delete the p1 object:

```
del p1
```

Unit – 5

Python File Handeling

File Handling

The key function for working with files in Python is the `open()` function.

The `open()` function takes two parameters; *filename*, and *mode*.

There are four different methods (modes) for opening a file:

`"r"` - Read - Default value. Opens a file for reading, error if the file does not exist

`"a"` - Append - Opens a file for appending, creates the file if it does not exist

`"w"` - Write - Opens a file for writing, creates the file if it does not exist

`"x"` - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

`"t"` - Text - Default value. Text mode

`"b"` - Binary - Binary mode (e.g. images)

Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

Because `"r"` for read, and `"t"` for text are the default values, you do not need to specify them.

Note: Make sure the file exists, or else you will get an error.

Open a File on the Server

Assume we have the following file, located in the same folder as Python:

demofile.txt

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

To open the file, use the built-in `open()` function.

The `open()` function returns a file object, which has a `read()` method for reading the content of the file:

Example

```
f = open("demofile.txt", "r")
print(f.read())
```

Read Only Parts of the File

By default the `read()` method returns the whole text, but you can also specify how many characters you want to return:

Example

Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Read Lines

You can return one line by using the `readline()` method:

Example

Read one line of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
```

By calling `readline()` two times, you can read the two first lines:

Example

Read two lines of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

By looping through the lines of the file, you can read the whole file, line by line:

Example

Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Close Files

It is a good practice to always close the file when you are done with it.

Example

Close the file when you are finish with it:

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

Note: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

Write to an Existing File

To write to an existing file, you must add a parameter to the `open()` function:

`"a"` - Append - will append to the end of the file

`"w"` - Write - will overwrite any existing content

Example

Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

Example

Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the appending:
f = open("demofile3.txt", "r")
print(f.read())
```

Note: the "w" method will overwrite the entire file.

Create a New File

To create a new file in Python, use the `open()` method, with one of the following parameters:

`"x"` - Create - will create a file, returns an error if the file exist

`"a"` - Append - will create a file if the specified file does not exist

`"w"` - Write - will create a file if the specified file does not exist

Example

Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Result: a new empty file is created!

Example

Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

Exception Handling

Python Try Except

The `try` block lets you test a block of code for errors.

The `except` block lets you handle the error.

The `finally` block lets you execute code, regardless of the result of the try- and except blocks.

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the `try` statement:

Example

The `try` block will generate an exception, because `x` is not defined:

```
try:
  print(x)
except:
  print("An exception occurred")
```

Since the try block raises an error, the except block will be executed.

Without the try block, the program will crash and raise an error:

Example

This statement will raise an error, because `x` is not defined:

```
print(x)
```

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example

Print one message if the try block raises a `NameError` and another for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

Else

You can use the `else` keyword to define a block of code to be executed if no errors were raised:

Example

In this example, the `try` block does not generate any error:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Finally

The `finally` block, if specified, will be executed regardless if the try block raises an error or not.

Example

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

This can be useful to close objects and clean up resources:

Example

Try to open and write to a file that is not writable:

```
try:
  f = open("demofile.txt")
  f.write("Lorum Ipsum")
except:
  print("Something went wrong when writing to the file")
finally:
  f.close()
```

The program can continue, without leaving the file object open.