Python Beginner Tutorials

Beginner

- Getting Started
- Numbers
- String basics
- String methods
- Lists
- Tuples
- Dictionaries
- Datatype casting
- If statements
- Functions
- Loops
- Random numbers
- Objects and classes
- Encapsulation
- Method overloading
- Inheritance
- Polymorphism
- Inner classes
- Factory method
- Binary numbers
- Recursive functions
- Logging
- Subprocess
- Threading

Top

Getting started

Python is a general-purpose computer programming language, ranked among the top eight most popular programming languages in the world.

It can be used to create many things including *web applications*, *desktop applications* as *scripting interpreter* and many more.

Please do note the online interpreters may not work for everything but will work for the beginner tutorials.

Run Python code online

- Skulpt Python interpreter
- Repl.it Python interpreter
- Ideone.com Python interpreter
- Codepad Python interpreter

Run Python code on your machine

- Official Python Installation Guide
- PyCharm IDE (recommended)

Try this code:
Try this code to test if Python is installed correctly.

```
#!/usr/bin/env python

print("Hello World!")
print("This is a Python program.")
```

(In Python 2.x you do not have to use the brackets around the print function, for Python 3.x is it required.)

Expected output:

```
Hello World!
This is a Python program
```

Next tutorial

Top

Python numbers

Python supports these data types for numbers:

name	purpose
int	whole number
long	long integers
float	floating point real values
complex	complex numbers

Example:

```
#!/usr/bin/python

x = 3                # an integer
f = 3.1415926        # a floating real point
name = "Python"      # a string
big = 358315791L     # long, a very large number
z = complex(2,3)     # (2+3i)  a complex number. consists of real an
d imaginary part.

print(x)
print(f)
print(name)
print(big)
print(z)
```

Output:

```
3
3.1415926
Python
358315791
(2+3j)
```

To find the maximum values depend on your platform.

The minimum and maximums on a ***32 bit machine***:

datatype	minimum	maximum
signed int	-2147483647	2147483647
long	–	limited only by memory
float	2.2250738585072014e-308	1.7976931348623157e+308

The number range on a ***64 bit machine:***

datatype	minimum	maximum
signed int	-9223372036854775807	9223372036854775807
long	–	limited only by memory
float 2.2250738585072014e-308		

datatype	minimum	maximum

Operations
You can do arithemetic operations such as addition (+), multiplication (*), division (/) and substractions (-).

```
#!/usr/bin/env python

x = 3
y = 8

sum = x + y

print(sum)
```

Expected output: 11.

User input
You can also ask the user for input using the *raw_input* function:

```
#!/usr/bin/env python
```

```
x = int(raw_input("Enter x:"))
y = int(raw_input("Enter y:"))

sum = x + y
print(sum)
```

In this case we want whole numbers (integers), which is why we write int() around the functions. If you want floating point numbers you would write float(raw_input("Enter x:")).

In the latest Python version you can use the input() function instead:

```
#!/usr/bin/env python

x = int(input("Enter x:"))
y = int(input("Enter y:"))

sum = x + y
print(sum)
```

Next tutorial (Strings) – Previous tutorial

Top

Python strings

If you use Python 3.x. put brackets around the print functions.

In Python we can do various operations on strings:

```
#!/usr/bin/python

s = "Hello Python"
print s        # prints whole string
print s[0]     # prints "H"
print s[0:2]   # prints "He"
print s[2:4]   # prints "ll"
print s[6:]    # prints "Python"
print s + ' ' + s # print concatenated string.
print s.replace('Hello','Thanks') # print a string with a replaced
 word
```

Output:

```
Hello Python
H
He
ll
Python
Hello Python Hello Python
Thanks Python
```

Python String compare

To compare two strings we can use the == operator.

```
#!/usr/bin/python

sentence = "The cat is brown"
q = "cat"

if q == sentence:
    print 'equal'
else:
    print 'not equal'
```

Python String contains

In Python you can test if a string contains a substring using this code:

```
#!/usr/bin/python

sentence = "The cat is brown"
q = "cat"

if q in sentence:
    print q + " found in " + sentence
```

Next tutorial (String methods)– Previous (numbers)

Top

String methods

You learned how to define strings, compare strings and test if the string contains something in the previous tutorial. In this article you will learn that there are more fun things you can do with strings.

Length of a string
We can get the length a string using the len() function.

```
#!/usr/bin/env python

s = "Hello world"   # define the string
print len(s)        # prints the length of the string
```

Output:

```
11
```

Converting to uppercase or lowercase
The function upper() can be called to convert a whole string to uppercase.

```
#!/usr/bin/env python

s = "Python"        # define the string
s = s.upper()       # convert string to uppercase
print s             # prints the string
```

Output:

```
PYTHON
```

Likewise we can convert a string to lower characters using the lower() function.

Adding strings together (concatinating)
We can add strings together use the plus operator:

```
#!/usr/bin/env python

str1 = "Python"
str2 = " is my favorite programming language."

print str1 + str2    # print a concatenated string.
```

Output:

```
Python is my favorite programming language.
```

Newlines in strings

In Python there are special characers that you can use in a string. You can use them to create newlines, tabs and so on. Let’s do example,we will use the “\n” or newline character:

```
#!/usr/bin/env python

str1 = "In Python,\nyou can use special characters in strings.\nTh
ese special characters can be..."
print str1
```

Output:

```
In Python,
you can use special characters in strings.
These special characters can be...
```

Quotes in strings

Sometimes you may want to show double quotes in the string, but because they are already used to start or end a string we have to escape them. An example:

```
#!/usr/bin/env python
```

```
str1 = "The word \"computer\" will be in quotes."
print str1
```

Output:

```
The word "computer" will be in quotes.
```

Special characters overview
An overview of special characters that you can use in strings:

Action	character
Newline	\n
Quotes	\”
Single quote	\’
Tab	\t
Backslash	\\

Next tutorial: Lists – Previous (strings)

Top

Python lists

Lists
A list can be used as:

```
#!/usr/bin/python

l = [ "Drake", "Derp", "Derek", "Dominique" ]

print l      # prints all elements
print l[0]  # print first element
print l[1]  # prints second element
```

Output:

```
['Drake', 'Derp', 'Derek', 'Dominique']
Drake
Derp
```

Adding and removing items
We can use the functions append() and remove() to manipulate the list.

```
#!/usr/bin/python

l = [ "Drake", "Derp", "Derek", "Dominique" ]

print l                     # prints all elements
l.append("Victoria")    # add element.
print l                     # print all elements
l.remove("Derp")          # remove element.
l.remove("Drake")         # remove element.
print l                     # print all elements.
```

Output:

```
['Drake', 'Derp', 'Derek', 'Dominique']
```

```
['Drake', 'Derp', 'Derek', 'Dominique', 'Victoria']
['Derek', 'Dominique', 'Victoria']
```

Sorting lists
We can sort the list using the sort() function.

```
#!/usr/bin/python

l = [ "Drake", "Derp", "Derek", "Dominique" ]

print l      # prints all elements
l.sort()     # sorts the list in alphabetical order
print l      # prints all elements
```

Output:

```
['Drake', 'Derp', 'Derek', 'Dominique']
['Derek', 'Derp', 'Dominique', 'Drake']
```

If you want to have the list in descending order, simply use the reverse() function.

```
#!/usr/bin/python

l = [ "Drake", "Derp", "Derek", "Dominique" ]

print l      # prints all elements
l.sort()     # sorts the list in alphabetical order
l.reverse() # reverse order.
print l      # prints all elements
```

Output:

```
['Drake', 'Derp', 'Derek', 'Dominique']
```

```
['Drake', 'Dominique', 'Derp', 'Derek']
```

Next tutorial (Tuples) – Previous (String methods)

Top

Python tuples

A tuple is a sequence of data. It is defined as a sequence of elements seperated by a comma.

```
#!/usr/bin/python

point = (3,4)
point2 = (2,6,12)

print point
print point[0]
print point[1]

print point2
print point2[0]
print point2[1]
```

Output:

```
(3, 4)
3
4
(2, 6, 12)
2
6
```

Next tutorial (Dictionaries) – Previous (Lists)

Top

Python dictionaries

A dictionary can be thought of as an unordered set of *key: value* pairs. A pair of braces creates an empty dictionary: {}. Each element can maps to a certain value. An integer or string can be used for the index. Dictonaries do not have an order. Let us make a simple dictionary:

```
#!/usr/bin/python

words = {}
words["Hello"] = "Bonjour"
words["Yes"] = "Oui"
words["No"] = "Non"
words["Bye"] = "Au Revoir"

print words["Hello"]
print words["No"]
```

Output:

```
Bonjour
Non
```

We are by no means limited to single word defintions in the value part. A demonstration:

```
#!/usr/bin/python

dict = {}
dict['Ford'] = "Car"
dict['Python'] = "The Python Programming Language"
dict[2] = "This sentence is stored here."

print dict['Ford']
print dict['Python']
print dict[2]
```

Output:

```
Car
The Python Programming Language
This sentence is stored here.
```

Manipulating the dictionary
We can manipulate the data stored in a dictionairy after declaration. This is shown in the example below:

```
#!/usr/bin/python

words = {}
words["Hello"] = "Bonjour"
words["Yes"] = "Oui"
words["No"] = "Non"
words["Bye"] = "Au Revoir"

print words              # print key-pairs.
del words["Yes"]         # delete a key-pair.
print words              # print key-pairs.
words["Yes"] = "Oui!"    # add new key-pair.
print words              # print key-pairs.
```

Output:

```
{'Yes': 'Oui', 'Bye': 'Au Revoir', 'Hello': 'Bonjour', 'No': 'Non'
}
{'Bye': 'Au Revoir', 'Hello': 'Bonjour', 'No': 'Non'}
{'Yes': 'Oui!', 'Bye': 'Au Revoir', 'Hello': 'Bonjour', 'No': 'Non
'}
```

Next tutorial (casting datatypes) – Previous (tuples)

Top

Datatype casting

To convert between datatypes you can use:

Function	Description
int(x)	Converts x to an integer
long(x)	Converts x to a long integer
float(x)	Converts x to a floating point number
str(x)	Converts x to an string. x can be of the type float. integer or long.
hex(x)	Converts x integer to a hexadecimal string
chr(x)	Converts x integer to a character
ord(x)	Converts character x to an integer

An example of casting datatypes in Python:
If you want to print numbers you will often need casting. In this example below we want to print two numbers, one whole number (integer) and one floating point number.

```
x = 3
y = 2.15315315313532

print "We have defined two numbers,"
print "x = " + str(x)
print "y = " + str(y)
```

Output:

```
We have defined two numbers,
x = 3
y = 2.15315315314
```

What if we have text that we want to store as number? We will have to cast again.

```
a = "135.31421"
b = "133.1112223"

c = float(a) + float(b)
```

```
print c
```

Output:

```
268.4254323
```

Next tutorial: Conditional statements– Previous (dictionary)

Top

Conditional statements

In Python you can define conditional statements, known as if-statements. Consider this application:

```
#!/usr/bin/python

x = 3
if x < 10:
   print 'x smaller than 10'
else:
   print 'x is bigger than 10 or equal'
```

Output:

```
x smaller than 10
```

If you set x to be larger than 10, it will execute the second code block.

A little game:
A variable may not always be defined by the user, consider this little game:

```
age = 24

print "Guess my age, you have 1 chances!"
guess = int(raw_input("Guess: "))

if guess != age:
    print "Wrong!"
else:
    print "Correct"
```

Conditional operators
A word on conditional operators

operator, description

!=, not equal

==, equals

>, greater than

Functions

Definition
A function is a set of reusable code that can be called from your program. You have used functions before including print()
They help you to structure the code and avoid repeating your code all over the place. The abstract structure is:

```
def function(parameters):
     instructions
     return value
```

We can call the function using function(parameters).

Example 1
Let us demonstrate that by an example:

```
#!/usr/bin/python

def f(x):
    return x*x

print f(3)
```

Output:

```
9
```

The function has one parameter, x. The return value is the value the function returns. Not all functions have to return something. You are by no means limited to simple arithmetic operations.

Example 2
We can pass multiple variables:

```
#!/usr/bin/python

def f(x,y):
```

```
    print 'You called f(x,y) with the value x = ' + str(x) + ' and
 y = ' + str(y)
    print 'x * y = ' + str(x*y)

f(3,2)
```

Output:

```
You called f(x,y) with the value x = 3 and y = 2
x * y = 6
```

Scope
Variables can only reach the area in which they are defined. This ***will not work***:

```
#!/usr/bin/python


def f(x,y):
    print 'You called f(x,y) with the value x = ' + str(x) + ' and
 y = ' + str(y)
    print 'x * y = ' + str(x*y)
    z = 4 # cannot reach z, so THIS WON'T WORK

z = 3
f(3,2)
```

but this will:

```
#!/usr/bin/python

def f(x,y):
    z = 3
    print 'You called f(x,y) with the value x = ' + str(x) + ' and
 y = ' + str(y)
    print 'x * y = ' + str(x*y)
    print z   # can reach because variable z is defined in the fun
```

```
ction

f(3,2)
```

Let’s examine this further:

```
#!/usr/bin/python

def f(x,y,z):
    return x+y+z    # this will return the sum because all variable
s are passed as parameters

sum = f(3,2,1)
print sum
```

We can also get the contents of a variable from another function:

```
#!/usr/bin/python

def highFive():
    return 5

def f(x,y):
    z = highFive()    # we get the variable contents from highFive
()
    return x+y+z      # returns x+y+z. z is reachable becaue it is
 defined above

result = f(3,2)
print result
```

A variable cannot be outside of the scope. The example below **does not work.**

```
#!/usr/bin/python

def doA():
```

```
    a = 5

def doB():
    print a    # does not know variable a, WILL NOT WORK!

doB()
```

but this example will:

```
#!/usr/bin/python

def doA():
    a = 5

def doB(a):
    print a    # we pass variable as parameter, this will work

doB(3)
```

In the last example we have two different variables named a, because the scope of the variable a is only within the function. That is to say, the variable is not known outside the scope. The function doB() will print number 3 (the value passed to it). The assignment a=5 is contained within function doA() and is never used and never visible to the function doB() or the rest of the code, except within the function doA() itself. To clarify, lets try to print a variable that is not known outside of the function:

```
#!/usr/bin/python

def doA():
    a = 5

print a    # does not work! a not defined.
```

If a variable can be reached anywhere in the code is called a ***global variable.*** If a variable is known only inside the scope, we call it a ***local variable.***

Next tutorial: Loops – Previous (if statement)

Top

Loops

In Python and many other programming languages you can repeat a part of code using a loop. A loop repeats a set of instructions N times. Python has 3 loops:

Type	Description
For	Executes the defined statements until the condition is met.
While	Executes the defined statements while a condition is true.
nested loops	Loops inside loops.

Python For loop example
We can iterate a list using a for loop

```
#!/usr/bin/python

items = [ "Abby","Brenda","Cindy","Diddy" ]

for item in items:
    print item
```

Output:

```
Abby
Brenda
Cindy
Diddy
```

The for loop can be used to repeat N times too:

```
#!/usr/bin/python

for i in range(1,10):
    print i
```

Output:

```
1
2
3
4
5
6
7
8
9
```

Python While loop example
Until a condition is met we can repeat some instructions. For example,

```
while button_not_pressed:
     drive()
```

Nested loops in Python:
We can combine for loops using nesting. If we want to iterate over an (x,y) field we could use:

```
#!/usr/bin/python

for x in range(1,10):
    for y in range(1,10):
        print "(" + str(x) + "," + str(y) + ")"
```

Output:

```
(1,1)
(1,2)
(1,3)
(1,4)
...
(9,9)
```

Nesting is very useful, but it increases complexity the deeper you nest.

Next tutorial: Random numbers – Previous (functions)

Top

Random numbers

Using the *random* module, we can generate pseudo-random numbers. The function random() generates a random number between zero and one [0, 0.1 .. 1]. Numbers generated with this module are not truly random but they are enough random for most purposes.

Random number between 0 and 1.
We can generate a (pseudo) random floating point number with this small code:

```
from random import *

print random()     # Generate a pseudo-
random number between 0 and 1.
```

Generate a random number between 1 and 100
To generate a whole number (integer) between one and one hundred use:

```
from random import *

print randint(1, 100)    # Pick a random number between 1 and 100.
```

This will print a random integer. If you want to store it in a variable you can use:

```
from random import *

x = randint(1, 100)    # Pick a random number between 1 and 100.
print x
```

Random number between 1 and 10
To generate a random *floating point number* between 1 and 10 you can use the uniform() function

```
from random import *

print uniform(1, 10)
```

Picking a random item from a list

Fun with lists
We can shuffle a list with this code:

```
from random import *

items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
shuffle(items)
print items
```

To pick a random number from a list:

```
from random import *

items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

x = sample(items,  1)   # Pick a random item from the list
print x[0]

y = sample(items, 4)    # Pick 4 random items from the list
print y
```

We can do the same thing with a list of strings:

```
from random import *

items = ['Alissa','Alice','Marco','Melissa','Sandra','Steve']

x = sample(items,  1)   # Pick a random item from the list
print x[0]

y = sample(items, 4)    # Pick 4 random items from the list
print y
```

Next tutorial: Objects and classes – Previous (loops)

Top

Objects and classes

In the very early days of computing, programmers wrote only instructions. That quickly became very complicated as there were hundreds of instructions. Functions helped to structure that code and it improved readability. Some time later a new paradigm was created: virtual objects or object orientated programming. In Python you can use this paradigm, so let us dive in depth.

Classes in Python
We can create virtual objects in Python. A virtual object can contain variables and methods. A program may have many different types and are created from a class. Consider this example:

```
class User:
    name = ""

    def __init__(self, name):
        self.name = name

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

# create virtual objects
james = User("James")
david = User("David")
eric = User("Eric")

# call methods owned by virtual objects
james.sayHello()
david.sayHello()
```

Run this program. In this code we have 3 virtual objects: james, david and eric. Each object is instance of the User class. In this class we defined the sayHello() method, which is why we can call it for each of the objects. The __init__() method is called the **constructor** and is always called when creating an object. The variables owned by the class is in this case “name”.

Exercises:

1. Add eric.sayHello() and run
2. Change the order and amount of sayHello() calls
3. Add a method sayBye() that displays “Goodbye”
4. Change the text inside User(“James”) and see what happens.

5. Create new virtual objects Brian, Tamara and Abbey and let them say hello.

If you have any questions feel free to drop a comment. Let us continue, we can create methods in classes which update the internal variables of the object. This may sound vague but I will demonstrate with an example.

Changing internal variables

We define a class CoffeeMachine of which the virtual objects hold the amount of beans and amount of water. Both are defined as a number (integer). We may then define methods that add or remove beans.

```
    def addBean(self):
        self.bean = self.bean + 1

    def removeBean(self):
        self.bean = self.bean - 1
```

We do the same for the variable water. As shown below:

```
class CoffeeMachine:
    name = ""
    beans = 0
    water = 0

    def __init__(self, name, beans, water):
        self.name = name
        self.beans = beans
        self.water = water

    def addBean(self):
        self.beans = self.beans + 1

    def removeBean(self):
        self.beans = self.beans - 1

    def addWater(self):
        self.water = self.water + 1

    def removeWater(self):
        self.water = self.water - 1

    def printState(self):
```

```
        print "Name  = " + self.name
        print "Beans = " + str(self.beans)
        print "Water = " + str(self.water)

pythonBean = CoffeeMachine("Python Bean", 83, 20)
pythonBean.printState()
print ""
pythonBean.addBean()
pythonBean.printState()
```

Run this program. The top of the code defines the class as we described. The code below is where we create virtual objects. In this example we have exactly one object called “pythonBean”. We then call methods which change the internal variables, this is possible because we defined those methods inside the class. Output:

```
Name  = Python Bean
Beans = 83
Water = 20

Name  = Python Bean
Beans = 84
Water = 20
```

Exercises:

1. Call addBean() multiple times
2. Change the name of machine
3. Create multiple coffee machines
4. Change the amount of water (don’t forget to print)
5. Create a class telephone with virtual objects Phone1 and Phone2 that can modify the amount of battery.

If you are stuck or have any questions, feel free to comment.

Next tutorial: Encapsulation – Previous (random)

Top

Encapsulation

In an object oriented python program, you can *restrict access* to methods and variables. This can prevent the data from being modified by accident and is known as *encapsulation*. Let’s start with an example.

Private methods
We create a class Car which has twomethods: drive() and updateSoftware(). When a car object is created, it will call the private methods __updateSoftware(). This function cannot be called on the object directly, only from within the class.

```
#!/usr/bin/env python

class Car:

    def __init__(self):
        self.__updateSoftware()

    def drive(self):
        print 'driving'

    def __updateSoftware(self):
        print 'updating software'

redcar = Car()
redcar.drive()
#redcar.__updateSoftware()  not accesible from object.
```

This program will output:

```
updating software
driving
```

The private method __updateSoftware() can only be called within the class itself. It can never be called from outside the class.

Private variables
Variables can be private which can be useful on many occasions. Objects can hold crucial data for your application and you do not want that data to be changeable from anywhere in

the code. An example:

```
#!/usr/bin/env python

class Car:

    __maxspeed = 0
    __name = ""

    def __init__(self):
        self.__maxspeed = 200
        self.__name = "Supercar"

    def drive(self):
        print 'driving. maxspeed ' + str(self.__maxspeed)

redcar = Car()
redcar.drive()
redcar.__maxspeed = 10  # will not change variable because its pri
vate
redcar.drive()
```

If you want to change the value of a private variable, a setter method is used. This is simply a method that sets the value of a private variable.

```
#!/usr/bin/env python

class Car:

    __maxspeed = 0
    __name = ""

    def __init__(self):
        self.__maxspeed = 200
        self.__name = "Supercar"

    def drive(self):
        print 'driving. maxspeed ' + str(self.__maxspeed)

    def setMaxSpeed(self,speed):
        self.__maxspeed = speed
```

```
redcar = Car()
redcar.drive()
redcar.setMaxSpeed(320)
redcar.drive()
```

Why would you create them? Because some of the private values you may want to change after creation of the object while others may not need to be changed at all.

Summary
To summarize, in Python there are:

Type	Description
public methods	accessible from anywhere
private methods	accessible only in their own class. starts with two underscores
public variables	accessible from anywhere
private variables	accesible only in their own class or by a method if defined. starts with two underscores

Other programming languages have protected class methods too, but Python does not.

Encapsulation gives you more control over the degree of coupling in your code, it allows a class to change its implementation without affecting other parts of the code.

Next tutorial: Method overloading – Previous (OOP)

Top

Method overloading

In Python you can define a method in such a way that there are multiple ways to call it. This is known as *method overloading*. We do that by setting default values of variables. Let us do an example:

```
#!/usr/bin/env python

class Human:

    def sayHello(self, name=None):

        if name is not None:
            print 'Hello ' + name
        else:
            print 'Hello '

# Create instance
obj = Human()

# Call the method
obj.sayHello()

# Call the method with a parameter
obj.sayHello('Guido')
```

Output:

```
Hello
Hello Guido
```

To clarify *method overloading*, we can now call the method sayHello() in two ways:

```
obj.sayHello()
obj.sayHello('Guido')
```

We created a method that can be called with fewer arguments than it is defined to allow.

We are not limited to two variables, your method could have more variables which are optional.

Next tutorial: Inheritance – Previous (encapsulation)

Top

Inheritance

Classes can inherit functionality from other classes, let’s take a look at how that works. We start with a basic class:

```
class User:
    name = ""

    def __init__(self, name):
        self.name = name

    def printName(self):
        print "Name  = " + self.name

brian = User("brian")
brian.printName()
```

This creates one instance called brian which outputs its given name. Add another class called Programmer.

```
class Programmer(User):

    def __init__(self, name):
        self.name = name
    def doPython(self):
        print "Programming Python"
```

This looks very much like a standard class except than User is given in the parameters. This means all functionality of the class User is accesible in the Programmer class.

Full example of Python inheritance:

```
class User:
    name = ""

    def __init__(self, name):
        self.name = name

    def printName(self):
```

```
        print "Name   = " + self.name

class Programmer(User):
    def __init__(self, name):
        self.name = name

    def doPython(self):
        print "Programming Python"

brian = User("brian")
brian.printName()

diana = Programmer("Diana")
diana.printName()
diana.doPython()
```

The output:

```
Name   = brian
Name   = Diana
Programming Python
```

Brian is an instance of User and can only access the method printName. Diana is an instance of Programmer, a class with inheritance from User, and can access both the methods in Programmer and User.

Next tutorial: Polymorphism – Previous (overloading)

Top

Polymorphism

Sometimes an object comes in many types or forms. If we have a button, there are many different draw outputs (round button, check button, square button, button with image) but they do share the same logic: onClick(). We access them using the same method . This idea is called *Polymorphism*.

Polymorphism is based on the greek words Poly (many) and morphism (forms). We will create a structure that can take or use many forms of objects.

Polymorphism in Python with a function:
We create two classes: Bear and Dog, both can make a distinct sound. We then make two instances and call their action using the same method.

```
class Bear(object):
    def sound(self):
        print "Groarrr"

class Dog(object):
    def sound(self):
        print "Woof woof!"

def makeSound(animalType):
    animalType.sound()


bearObj = Bear()
dogObj = Dog()

makeSound(bearObj)
makeSound(dogObj)
```

Output:

```
Groarrr
Woof woof!
```

Polymorphism with abstract class (most commonly used)
If you create an editor you may not know in advance what type of documents a user will

open (pdf format or word format?). Wouldn't it be great to acess them like this, instead of having 20 types for every document?

```
for document in documents:
    print document.name + ': ' + document.show()
```

To do so, we create an abstract class called document. This class does not have any implementation but defines the structure (in form of functions) that all forms must have. If we define the function show() then both the PdfDocument and WordDocument must have the show() function. Full code:

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

    def show(self):
        raise NotImplementedError("Subclass must implement abstrac
t method")

class Pdf(Document):
    def show(self):
        return 'Show pdf contents!'

class Word(Document):
    def show(self):
        return 'Show word contents!'

documents = [Pdf('Document1'),
             Pdf('Document2'),
             Word('Document3')]

for document in documents:
    print document.name + ': ' + document.show()
```

Output:

```
Document1: Show pdf contents!
Document2: Show pdf contents!
Document3: Show word contents!
```

We have an abstract access point (document) to many types of objects (pdf,word) that follow the same structure.

Another example would be to have an abstract class Car which holds the structure drive() and stop(). We define two objects Sportscar and Truck, both are a form of Car. In pseudo code what we will do is:

```
class Car:
    def drive abstract, no implementation.
    def stop abstract, no implementation.

class Sportscar(Car):
    def drive: implementation of sportscar
    def stop: implementation of sportscar

class Truck(Car):
    def drive: implementation of truck
    def stop: implementation of truck
```

Then we can access any type of car and call the functionality without taking further into account if the form is Sportscar or Truck. Full code:

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

    def drive(self):
        raise NotImplementedError("Subclass must implement abstrac
t method")

    def stop(self):
        raise NotImplementedError("Subclass must implement abstrac
t method")

class Sportscar(Car):
    def drive(self):
        return 'Sportscar driving!'

    def stop(self):
        return 'Sportscar breaking!'
```

```
class Truck(Car):
    def drive(self):
        return 'Truck driving slowly because heavily loaded.'

    def stop(self):
        return 'Truck breaking!'


cars = [Truck('Bananatruck'),
        Truck('Orangetruck'),
        Sportscar('Z3')]

for car in cars:
    print car.name + ': ' + car.drive()
```

Output:

```
Bananatruck: Truck driving slowly because heavily loaded.
Orangetruck: Truck driving slowly because heavily loaded.
Z3: Sportscar driving!
```

Next tutorial: Inner classes – Previous (inheritance)

Top

Inner classes

An **inner class** or **nested class** is a defined entirely within the body of another class Let us do an example:

```
#!/usr/bin/env python

class Human:

 def __init__(self):
    self.name = 'Guido'
    self.head = self.Head()

 class Head:
    def talk(self):
    return 'talking...'

if __name__ == '__main__':
    guido = Human()
    print guido.name
    print guido.head.talk()
```

Output:

```
Guido
talking...
```

In the program above we have the inner class Head() which has its own method. An inner class can have both methods and variables. In this example the constructor of the class Human (__init__) creates a new head object. You are by no means limited to the number of inner classes, for example this code will work too:

```
#!/usr/bin/env python

class Human:

  def __init__(self):
    self.name = 'Guido'
    self.head = self.Head()
```

```
    self.brain = self.Brain()

  class Head:
    def talk(self):
      return 'talking...'

  class Brain:
    def think(self):
      return 'thinking...'

if __name__ == '__main__':
  guido = Human()
  print guido.name
  print guido.head.talk()
  print guido.brain.think()
```

By using inner classes you can make your code even more object orientated. A single object can hold several sub objects. We can use them to add more structure to our programs.

Next tutorial: Factory method – Previous (polymorphism)

Top

Factory method

We may not always know what kind of objects we want to create in advance. Some objects could be created *only* at execution time after a user requests so. Examples: A user may click on a certain button that creates an object. A user may create several new documents of different types. If a user starts a webbrowser, the browser does not know in advance how many tabs (where every tab is an object) will be opened.

To deal with this we can use the *factory method* pattern. The idea is to have one function, the factory, that takes an input string and outputs an object. Thus, the factory returns objects.

```
obj = Car.factory("Racecar")
obj.drive()
```

The type of object depends on the type of input string you specify. This technique could make your program more easily extensible also. A new programmer could easily add functionality by adding a new string and class, without having to read all of the source code.

Full code:

```
class Car(object):

    def factory(type):
        if type == "Racecar":
            return Racecar()
        if type == "Van":
            return Van()
        assert 0, "Bad car creation: " + type

    factory = staticmethod(factory)

class Racecar(Car):
    def drive(self): print("Racecar driving.")

class Van(Car):
    def drive(self): print("Van driving.")

# Create object using factory.
obj = Car.factory("Racecar")
obj.drive()
```

Output:

```
Racecar driving.
```

Next tutorial: Binary numbers and operations – Previous (inner class)

Top

Binary numbers and logical operators

We have looked at simple numbers and operations before. In this article you will learn how numbers work inside the computer and a some of magic to go along with that

More detailed: While this is not directly useful in web applications or most desktop applications, it is very useful to know. If you are only interested in that, you can skip to one of the next tutorials. In this article you will learn how to use binary numbers in Python, how to convert them to decimals and how to do bitwise operations on them.

Binary numbers
At the lowest level, the computer has no notion whatsoever of numbers except ‘there is a signal’ or ‘these is not a signal’. You can think of this as a light switch: Either the switch is on or it is off.

This tiny amount of information, the smallest amount of information that you can store in a computer, is known as a *bit.* We represent a bit as either low (0) or high (1).

To represent higher numbers than 1, the idea was born to use a sequence of bits. A sequence of eight bits could store much larger numbers, this is called a *byte*. A sequence consisting of ones and zeroes is known as *binary*. Our traditional counting system with ten digits is known as decimal.

Binary	0	1
Decimal	0	1

Binary	0 0	0 1	1 0	1 1
Decimal	0	1	2	3

Binary numbers and their decimal representation.

Lets see that in practice:

```
# Prints out a few binary numbers.
print int('00', 2)
print int('01', 2)
print int('10', 2)
print int('11', 2)
```

The second parameter 2, tells Python we have a number based on 2 elements (1 and 0). To convert a byte (8 bits) to decimal, simple write a combination of eight bits in the first parameter.

```
# Prints out a few binary numbers.
print int('00000010', 2)   # outputs 2
print int('00000011', 2)   # outputs 3
print int('00010001', 2)   # outputs 17
print int('11111111', 2)   # outputs 255
```

How does the computer do this? Every digit (from right to left) is multiplied by the power of two.

The number ‘000**1**000**1**‘ is (**1** x 2^0) + (0 x 2^1) + (0 x 2^2) + (0 x 2^3) + (**1** x 2^4) + (0 x 2^5) + (0 x 2^6) + (0 x 2^7) = 16 + 1 = 17. Remember, read from right to left.

The number ‘00110010’ would be (0 x 2^0) + (1 x 2^1) + (0 x 2^2) + (0 x 2^3) + (**1** x 2^4) + (**1** x 2^5) + (0 x 2^6) + (0 x 2^7) = 32+16+2 = 50.

Try the sequence ‘00101010’ yourself to see if you understand and verify with a Python program.

Logical operations with binary numbers

Binary Left Shift and Binary Right Shift
Multiplication by a factor two and division by a factor of two is very easy in binary. We simply shift the bits left or right. We shift left below:

Bit 4	Bit 3	Bit2	Bit 1
0	1	0	1
1	0	1	0

Before shifting (0,1,0,1) we have the number 5 . After shifting (1,0,1,0) we have the number 10. In python you can use the bitwise left operator (<<) to shift left and the bitwise right operator (>>) to shift right.

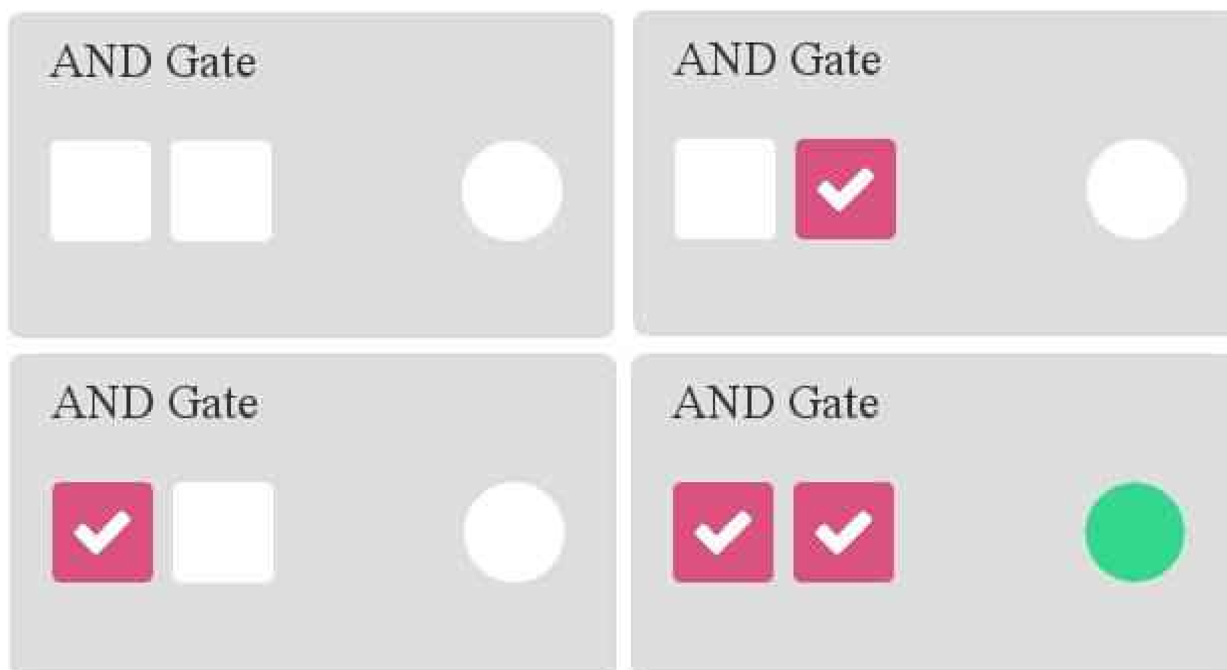
```
inputA = int('0101',2)

print "Before shifting " + str(inputA) + " " + bin(inputA)
print "After shifting in binary: " + bin(inputA << 1)
print "After shifting in decimal: " + str(inputA << 1)
```

Output:

```
Before shifting 5 0b101
After shifting in binary: 0b1010
After shifting in decimal: 10
```

The AND operator
Given two inputs, the computer can do several logic operations with those bits. Let’s take the AND operator. If input A and input B are positive, the output will be positive. We will demonstrate the AND operator graphically, the two left ones are input A and input B, the right circle is the output:



Bitwise AND

In code this is as simple as using the & symbol, which represents the Logical AND operator.

```
# This code will execute a bitwise logical AND. Both inputA and in
putB are bits.
inputA = 1
inputB = 1
print inputA & inputB   # Bitwise AND
```

By changing the inputs you will have the same results as the image above. We can do the AND operator on a sequence:

```
inputA = int('00100011',2)   # define binary sequence inputA
inputB = int('00101101',2)   # define binary sequence inputB

print bin(inputA & inputB)   # logical AND on inputA and inputB an
d output in binary
```

Output:

```
0b100001   # equals 00100001
```

This makes sense because if you do the operation by hand:

```
00100011
00101101
--------   Logical bitwise AND
00100001
```

The OR operator
Now that you have learned the AND operator, let’s have a look at the OR operator. Given two inputs, the output will be zero only if A and B are both zero.

binary bitwise OR

To execute it, we use the | operator. A sequence of bits can simply be executed like this:

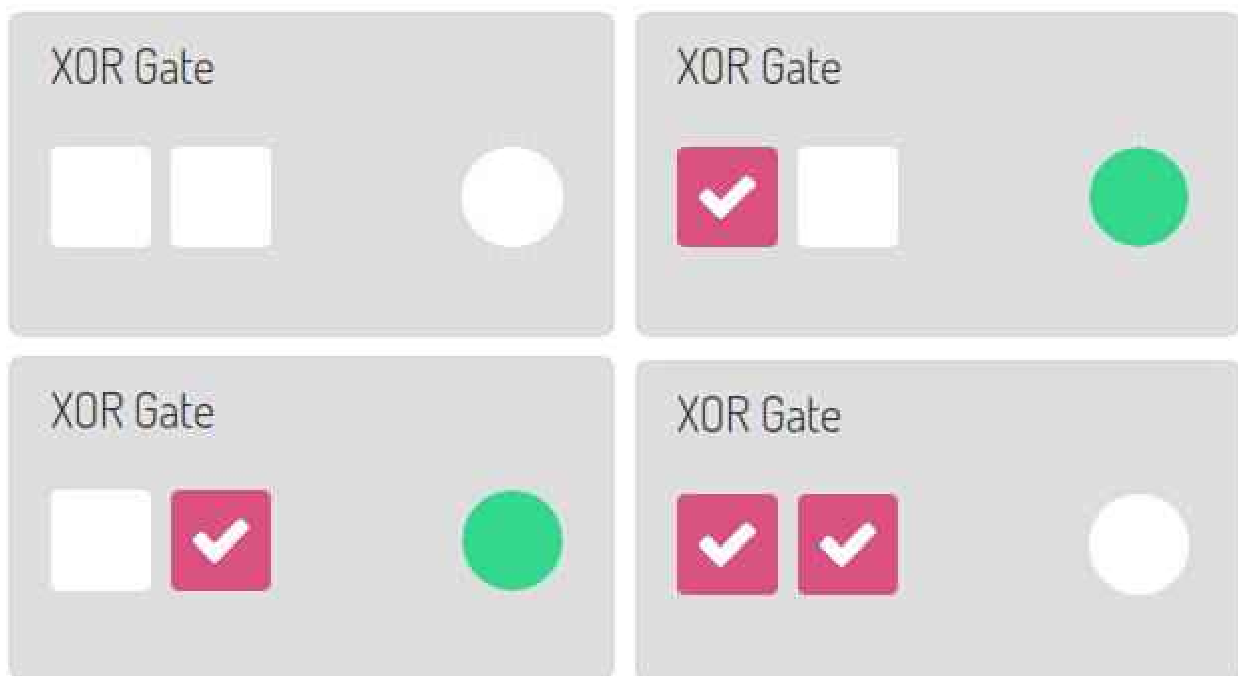
```
inputA = int('00100011',2)  # define binary number
inputB = int('00101101',2)  # define binary number

print bin(inputA)            # prints inputA in binary
print bin(inputB)            # prints inputB in binary
print bin(inputA | inputB)   # Execute bitwise logical OR and prin
t result in binary
```

Output:

```
0b100011
0b101101
0b101111
```

The XOR operator
This is an interesting operator: The Exclusive OR or shortly XOR.



bitwise XOR

To execute it, we use the ^ operator. A sequence of bits can simply be executed like this:

```
inputA = int('00100011',2)  # define binary number
inputB = int('00101101',2)  # define binary number

print bin(inputA)             # prints inputA in binary
print bin(inputB)             # prints inputB in binary
print bin(inputA ^ inputB)    # Execute bitwise logical OR and prin
```

```
t result in binary
```

Output:

```
0b100011
0b101101
0b1110
```

Next page: Recursive functions – Previous (factory method)

Top

Recursion

In English there are various examples of recursion: “To understand recursion, you must first understand recursion”, “A human is someone whose mother is human”. You might wonder, what does this have to do with programming?

You may want to split a complex problem into several smaller ones. You are already familiar with loops or iterations, but in some situations recursion may be a better solution. In Python, a function is said to be recursive if it calls itself and has a termination condition. Why a termination condition? To stop the function from calling itself ad infinity.

Recursion in with a list
Let’s start with a very basic example: adding all numbers in a list. Without recursion, this could be:

```
#!/usr/bin/env python

def sum(list):
    sum = 0

    # Add every number in the list.
    for i in range(0, len(list)):
        sum = sum + list[i]

    # Return the sum.
    return sum

print(sum([5,7,3,8,10]))
```

Where we simply call the sum function, the function adds every element to the variable sum and returns. To do this recursively:

```
#!/usr/bin/env python

def sum(list):
   if len(list) == 1:
        return list[0]
   else:
        return list[0] + sum(list[1:])

print(sum([5,7,3,8,10]))
```

If the length of the list is one it returns the list (the termination condition). Else, it returns the element and a call to the function sum() minus one element of the list. If all calls are executed, it returns reaches the termination condition and returns the answer.

Factorial with recursion
The mathematical definition of factorial is: n! = n * (n-1)!, if n > 1 and f(1) = 1. Example: 3! = 3 x 2 x 1 = 6. We can implement this in Python using a recursive function:

```
#!/usr/bin/env python

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

print factorial(3)
```

When calling the factorial function n = 3. Thus it returns n * factorial(n-1). This process will continue until n = 1. If n==1 is reached, it will return the result.

Limitations of recursions
Everytime a function calls itself and stores some memory. Thus, a recursive function could hold much more memory than a traditional function. Python stops the function calls after a depth of 1000 calls. If you run this example:

```
#!/usr/bin/env python

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

print factorial(3000)
```

You will get the error:

```
RuntimeError: maximum recursion depth exceeded
```

In other programming languages, your program could simply crash. You can resolve this by modifying the number of recursion calls such as:

```
#!/usr/bin/env python
import sys

sys.setrecursionlimit(5000)

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

print factorial(3000)
```

but keep in mind there is still a limit to the input for the factorial function. For this reason, you should use recursion wisely. As you learned now for the factorial problem, a recursive function is not the best solution. For other problems such as traversing a directory, recursion may be a good solution.

Next tutorial: Logging – Previous (binary)

Top

Logging

Python logging

We can track events in a software application, this is known as **logging**. Let’s start with a simple example, we will log a warning message:

```
import logging

# print a log message to the console.
logging.warning('This is a warning!')
```

This will output:

```
WARNING:root:This is a warning!
```

We can easily output to a file:

```
import logging

logging.basicConfig(filename='program.log',level=logging.DEBUG)
logging.warning('An example message.')
logging.warning('Another message')
```

The importance of a log message depends on the severity.

Level of severity

The logger module has several levels of severity. We set the level of severity using this line of code:

```
logging.basicConfig(level=logging.DEBUG)
```

These are the levels of severity:

Type	Description

Type	Description
DEBUG	Information only for problem diagnostics
INFO	The program is running as expected
WARNING	Indicate something went wrong
ERROR	The software will no longer be able to function
CRITICAL	Very serious eror

The default logging level is warning, which implies that other messages are ignored. If you want to print debug or info log messages you have to change the logging level like so:

```
import logging

logging.basicConfig(level=logging.DEBUG)
logging.debug('Debug message')
```

Time in log
You can enable time for logging using this line of code:

```
logging.basicConfig(format='%(asctime)s %(message)s')
```

An example below:

```
import logging

logging.basicConfig(format='%(asctime)s %(message)s', level=loggin
g.DEBUG)
logging.info('Logging app started')
logging.warning('An example logging message.')
logging.warning('Another log message')
```

Output:

```
2015-06-25 23:24:01,153 Logging app started
2015-06-25 23:24:01,153 An example message.
2015-06-25 23:24:01,153 Another message
```

Next tutorial: Subprocess – Previous (recursion)

Top

Python Subprocess

The subprocess module enables you to start new applications from your Python program. How cool is that?

Starting a process in Python:
You can start a process in Python using the Popen function call. The program below starts the unix program 'cat' and the second parameter is the argument. This is equivalent to 'cat test.py'. You can start any program with any parameter.

```
#!/usr/bin/env python

from subprocess import Popen, PIPE

process = Popen(['cat', 'test.py'], stdout=PIPE, stderr=PIPE)
stdout, stderr = process.communicate()
print stdout
```

The process.communicate() call reads input and output from the process. stdout is the process output. stderr will be written only if an error occurs. If you want to wait for the program to finish you can call Popen.wait().

Another method to start a process in Python:
Subprocess has a method call() which can be used to start a program. The parameter is a list of which the first argument must be the program name. The full definition is:

```
subprocess.call(args, *, stdin=None, stdout=None, stderr=None, she
ll=False)
# Run the command described by args.
# Wait for command to complete, then return the returncode attribu
te.
```

In the example below the full command would be "ls -l"

```
#!/usr/bin/env python

import subprocess
subprocess.call(["ls", "-l"])
```

Running a process and saving to a string
We can get the output of a program and store it in a string directly using check_output. The method is defined as:

```
 subprocess.check_output(args, *, stdin=None, stderr=None, shell=F
alse, universal_newlines=False)
 # Run command with arguments and return its output as a byte stri
ng.
```

Example usage:

```
#!/usr/bin/env python
import subprocess

s = subprocess.check_output(["echo", "Hello World!"])
print("s = " + s)
```

Next tutorial: Threading – Previous (logging)

Top

Threading

In Python you can create threads using the thread module in Python 2.x or _thread module in Python 3. We will use the threading module to interact with it.

A thread is an operating system process with different features than a normal process:

- threads exist as a subset of a process
- threads share memory and resources
- processes have a different address space (in memory)

When would you use threading? Usually when you want a function to occur at the same time as your program. If you create server software, you want the server not only listens to one connection but to many connections. In short, threads enable programs to execute multiple tasks at once.

Python threading
Let’s create a thread program. In this program we will start 10 threads which will each output their id.

```
import threading

# Our thread class
class MyThread (threading.Thread):

    def __init__(self,x):
        self.__x = x
        threading.Thread.__init__(self)

    def run (self):
          print str(self.__x)

# Start 10 threads.
for x in xrange(10):
    MyThread(x).start()
```

Output:

```
0
1
...
9
```

Threads do not have to stop if run once. Threads could be timed, where a threads functionality is repeated every x seconds.

Timed threads

In Python, the Timer class is a subclass of the Thread class. This means it behaves similar. We can use the timer class to create timed threads. Timers are started with the .start() method call, just like regular threads. The program below creates a thread that starts after 5 seconds.

```
#!/usr/bin/env python
from threading import *

def hello():
    print "hello, world"

# create thread
t = Timer(10.0, hello)

# start thread after 10 seconds
t.start()
```

Repeating functionality using threads

We can execute threads endlessly like this:

```
#!/usr/bin/env python
from threading import *
import time

def handleClient1():
    while(True):
        print "Waiting for client 1..."
        time.sleep(5) # wait 5 seconds

def handleClient2():
    while(True):
        print "Waiting for client 2..."
        time.sleep(5) # wait 5 seconds

# create threads
t = Timer(5.0, handleClient1)
```

```
t2 = Timer(3.0, handleClient2)

# start threads
t.start()
t2.start()
```

Top