Python for JNTUK Students

Sreekanth Kolamala

M.tech Computer Science, IIT Kanpur

unit 1

History of python

Python was created by Guido van Rossum in 1991.

Need of python programming

C, C++ are powerful langauges. langauges that can be used to write a variaty of software. they give you control of the machine and the very minitue control the system. But, they are very low level. we need to specify very small detail. For example, we have to create and manage memory requirement, object destruction etc, when all we need to do is save a few objects.

C, C++ syntax makes writing code very lengthly. C, C++ focus on reducing memory, running programs fast etc. Python's focus is to reduce developer time-time taken by the programmer to create software. C's 20 line code would be 5 or less lines long.

Applications

Python's concise code makes it real quick to write software. It's high level. Python's is used for the real software that we see often in our day-to-day life.

web development

Django is popular python framework(a big libraby is called a framework) that simplifies build websites like amazon or flipkart.

Data science and maching learning

Data science is different from the general programming. In general programming, finding out the average of subjects marks is a typical programming. Data science trys to predict the answer to question 'when will you come to college tommorow?' It does it by observing previous data. Since you came on 8:30 everyday for the past 6 months, you will come tommorow at 8:30. This is simple problem but typical problems in this field are like 'what is chance of getting cancer if you have a family history?' 'what will the whether be today?' etc. Two langauges dominate this field - Python and R. Both of which are in your sylabus.

Gaming and gui applications

The famous game Civiliztion IV where you get build an empire was written in python.

Internet of Things(IoT)

Our calculators are computers. Not as powerful as our laptops but still are computers. Iot is a technology where, small calculator sized but more powerful computers are attached everyday devices like TV, Fridge, AC to make them smart. The smart house where you can voice operate lights, doors, music is becuase of iot. And, python is the primary language of this field.

Natural language processing

Did you try giving voice commands to your smart phone? Isn't it cool? the phone listens and finds out what you are talking out. It does this even if an acsent is present. This field of find out what humans are saying is called natural language processing. Python has cool librabies for this too.

Setup

Download python from here: https://www.python.org/downloads/.

JNTUK does not specific which version to download. Python has two main versions. Python 2 and python 3. Download python 2. At the time of writing this book(July 2017), the latest verision was 2.7.13. Once you download the exe file, just double click and install. This should be very easily.

Click on windows start and find python. In windows 7 and above, you could just type in python and in the search results something like **python(command line)** will come up. Click on that, and a black screen will come. That is the **python interpreter**. That is where we will run all our python code. We will open this black screen each time we need to do some coding, so become comfortable opening up this application.

REPL

Open the **python interpreter** and type in the following one line at a time:

```
print 2 + 3
print 'sreekanth'
45 + 21.1
45/9
```

```
Out: 5
Out: 66.1
Out: 5
```

Each line you typed above is a command for the python interpreter. It reads your command, evaluates it and prints the result. when you typed 2 + 3, python took this command, calculated the sum and then printed it. Again the same steps of reading, evaluating, printing happened for the second command. This loop is called the **read-eval-print** loop. And the python interpreter is also called the **python repl**. The nice thing about python repl is it works as a calculator as seen above.

you can even type in a complex expression

```
print 'sree' in 'sreekanth'
```

```
Out: True
```

Don't worry what this expression does. For now, just remember you can type **any** python expression.

you can type any python expression

Running python scripts

save the previous code listing into a file and name it script.py. run that file using the command

```
run script.py
```

All python files should end with '.py'. you can give then any name(following the rules of variable naming).

Variables and Assignment

In programming, we often need to save some number, some information somewhere so that we can use it later. Variables are used to save such infomation.

```
age = 32
name = 'sreekanth'
```

Here, `age` is variable to which the infomation 32 is saved. `name` is another variable to which the infomation 'sreekanth' is saved. we can see what infomation a particular variable has simply by printing them(don't worry about the quotes, we will get to that when we learn about strings).

```
print age
print name
```

```
Out: 32
Out: sreekanth
```

We could also modify the infomation assigned to a variable.

```
age = age + 10
print age
name = name + 'kolamala'
print name
```

```
Out: 42
```

Here, we are assigning the variable age, the previously saved age and a 10 more. So, now the variable age would have the infomation 42.

If you are familar with C or java, you would noticed the is not variable declaration like `int age = 32;`. In python, you don't declare variables before assigning some infomation. you just use them. Also, the semicolon';' is optional in python-you can use it but is not required.

Keywords

Each langauge has a set of words that are reserved. They can't be used as variable names. The complete list is given here as referance. you don't need to know them by heart. This is just for your reference.

False	class	finally	is	return
in	continue	for	lambda	try
True	def	from	raise	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except			

Input-output

```
print 'some-string'
print 23
# to print multiple variables on the same line:
print 'some-sting', 23, 45.5
```

```
Out: some-string
Out: 23
Out: some-sting 23 45.5
```

```
# Input
a = raw_input("enter a number:")
```

`raw_input()` always takes the input as string. Even if you type numbers, they would be taken as string. So, if you really want to input numbers, you need to convert them. We use the built-in python functions `int()` , `float()` for this.

```
a = raw_input("enter a number:")
print a
print type(a)
a = int(a)
print type(a)
```

```
Out: enter a number:23
Out: 23
Out: <type 'str'>
Out: <type 'int'>
```

Python programmers combine the input and conversion into one line like this:

```
age = int(raw_input('enter age'))
height = float(raw_input('enter height'))
```

Indentation

covered in if section

```
print 'this line will be printed'
  print 'this will cause indentation error'
```

```
Out:   File "<string>", line 2
Out:     print 'this will cause indentation error'
Out:     ^
Out: IndentationError: unexpected indent
```

We will talk about indentation more when we talk about if statements. But, for now remember to indent all the statements uniformly.

unit 2

Types

Python has 5 data types-Numbers, Strings, Lists, Tuple, Dictionary. In this chapter, we will discuss Numbers and Strings. We will see the other 3 in the next chapter.

Numbers

Numbers are further divided into 4:int, long, float, complex. int and long are used to represent whole numbers. float is used for numbers that have a decimal part. Languages like C have double type for really big floating point numbers. But python only has float.

```
a = 12
print type(a)
#type is a built-in function that returns the type of a variable
b = 12.3
print type(b)
c = 2323231211142342424
print type(c)
d = complex(2,3)
print d
print type(d)
```

```
Out: <type 'int'>
Out: <type 'float'>
Out: <type 'long'>
Out: (2+3j)
Out: <type 'complex'>
```

The data type complex represent mathematical complex numbers that have a real and an imaginary part.

Strings

Python doesn't differete between single quotes and double quotes. Unlike in C, the is not difference between char ans string-both are same.

```
name = "sreekanth"
occupation = 'software engineer'
print type(name)
qoute = "I'am good" #observe that a single qoute can be used within double quotes
print "qoute:" + qoute
qoute = "qoute:" + 'He said "I am not going" to her'
print qoute
```

```
Out: <type 'str'>
Out: qoute:I'am good
Out: qoute:He said "I am not going" to her
```

```
qoute = "He said "I am not going" to her" #this will not work.
```

Booleans

Boolean variables can be assigned either `True` or `False`

```
flag = True
print flag
flag = False
print flag
```

```
Out: True
Out: False
```

Operators

Variables are assigned values using the = operator. Variables are written when they are first assigned a value.

```
number = 10
```

The variable `number` is assigned 10. Note that the type of variable `number` is not declared. This is not required because of the python type inference. Based on the operations we do on a variable, the python will automatic guess its type.

```
#multiple assignments can be done in the same statement simultaneously.
a, b = 10,20
#a is assigned 10 and b is assigned 20.
print a,b
a,b = b,a
#a is assigned the value in b and b is assigned the value in a simultaneously.
#this is a popular way of swapping numbers in python.
print a,b
```

```
Out: 10 20
Out: 20 10
```

Arithmetic Operators

They are `+ - * / % **`. `a%b` is gives the reminder of when a is divided by b. `a**b` gives a to power of b.

operator	example	result
+	2 + 3	5
-	2.3 - 3.5	-1.2
*	2*1.5	3
/	5/2	2
%	5%2	1
**	5**2	25

Relational Operators

operator	example	result
>	20 > 40	False
	20 > 20	False
>=	20 >= 20	True
<	20 < 40	True
	20 < 20	False
<=	20 <= 20	True
==	20 == 20	True
	20 == 40	False
!=	20 != 20	False
	20 != 40	True

Logical Operators

There are three logical operators: not, or, and. Also, >, >=, <, <=, ==, !=

```
condition = True
print (not condition)
print (condition or False)
print (condition and False)
```

```
Out: False
Out: True
Out: False
```

`not a` evaluates to the opposite of what `a` is. `a and b` evaluates to True only if both a, b are True. a or b evaluates to False if both a,b are False. They behave exactly like !, ||, && operators in C.

a	not a
True	False
False	True

a	b	a and b
True	True	True
True	False	False
False	True	False
False	False	False

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

Membership Operator

The `in` operator is used to determine variable consists of other.

```
isPresent = "sree" in "sreekanth"
print isPresent
```

```
Out: True
```

We will discuss this operator in more detail in unit-3 once we complete lists.

```
fruits = ["apple", "banana", "orange"]
print "apple" in fruits
#the 'in' operator can be used with the 'not' operator
#to find out if a variable is not present in another
print "banana" not in fruits
```

```
Out: True
Out: False
```

Terary operator

a if condition else b

```
number = 24
print "odd" if number % 2 else "even"
```

```
Out: even
```

Control flow

```
a,b = 3,2
if(a > b):
  print "a is greater than b" #Notice this print stmt is indented
  #this means that the print stmt is within the if block
```

```
Out: a is greater than b
```

Python seperates blocks based on there indentation. Notice the priNotice unlike other langauges, python doesn't use curly brackets{} to block. The above if statement can also be written as if-else statement.

```
a,b = 10,20
if a > b:
  print "a is greater than b"
else:
  pass
```

pass is a keyword that represents an empty statement. It's is like an empty ; in C.

```
a,b = 3,2
if a > b:
  print "this statement is within the if block"
  print "this statement aswell"
else:
  print "this statement is in the else block"
  print "this statement is also in the else block"
print "this statement is not in the if block. it's in the global block and will always print"
```

```
Out: this statement is within the if block
Out: this statement aswell
Out: this statement is not in the if block. it's in the global block and will always print
```

Loops

The for loop

```
numbers = range(1,10)
for number in numbers:
  print number
```

```
Out: 1
Out: 2
Out: 3
Out: 4
Out: 5
```

```
Out: 6
Out: 7
Out: 8
Out: 9
```

```
for aphlabet in "sreekanth":
  print aphlabet
```

```
Out: s
Out: r
Out: e
Out: e
Out: k
Out: a
Out: n
Out: t
Out: h
```

```
fruits = ["apple", "banana", "orange"]
while len(fruits) != 0:
  print fruits.pop()
print fruits
```

```
Out: orange
Out: banana
Out: apple
Out: []
```

```
numbers = range(0,10) #remember range() creates a list [0,1,2...,9]
#program to print even numbers
for i in numbers:
  if i%2 != 0:
    continue
  print i #remember indentation rules: this stmt is in for block but not in if block
print "program end"
```

```
Out: 0
Out: 2
Out: 4
Out: 6
Out: 8
Out: program end
```

Each time the keyword `continue` is hit, the remaining part of the for loop is not executed

```
numbers = range(0,10)
for i in numbers:
  if i%2 != 0:
    break
  print i
print "program end"
```

```
Out: 0
Out: program end
```

Notice that this program produces nothing. When `i=1`, the `if` condition is True, and break statement is hit. When a break statement is hit, the loop, the break statement is in, is immediately exited.

Lists

Lists are similar to C arrays but more powerful. Lists store multiple values. unlike C arrays, elements in a python list don't have be of the same type. lists are indexed from 0.

```
fruits = ['apple', 'mango', 'pineapple']
numbers = [10, 20, 30, 40, 50]
#elements in a list can be accessed using index.
print numbers[1]
#even negative index is allowed, in which case length of list is added
#to the index.
print numbers[-1]
fruitsandnumbers =  ['apple', 'mango', 'pineapple', 10, 20, 30, True]
#mixing elements of different types is fine
print fruitsandnumbers
print len(numbers)
#len is a built-in function used to find the length of lists.
integers = range(1,10)
print integers
#range(start,last) is built-in function that creates a list [start, start+1, ..., last-1]
```

```
Out: 20
Out: 50
Out: ['apple', 'mango', 'pineapple', 10, 20, 30, True]
Out: 5
Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Slicing

Slicing is a simple way of creating a new list from (part of) an old list.

newList = oldList[start:end]

```
numbers = [10, 20, 30, 40, 50, 60, 70]
sublist = numbers[1:4]
print sublist
print numbers[0:3]
print numbers[:3] #missing starting number, with replaced with 0
print numbers[4:7]
print numbers[4:] #missing ending number is replaced with length of the list.
#negative indices can also be given.
#In which case, the length of the list is added
print numbers[-2:7]
print numbers[:-1]  #this will print all elements except the last one.
```

```
Out: [20, 30, 40]
Out: [10, 20, 30]
Out: [10, 20, 30]
Out: [50, 60, 70]
Out: [50, 60, 70]
Out: [60, 70]
Out: [10, 20, 30, 40, 50, 60]
```

As shown in the example, sublist is a new list consisting of elements 1 to 3(remember, 'end' is not inclusive) in the numbers list. you can also specify negative numbers for indexes in which case, the length of list is

automatically added. For example `numbers[-2:7]` is equivalent to `numbers[5:7]`.

List methods

Lists are not fixed in length. They increase and decrease when elements are added and removed.

```
numbers = [10,20,30,40,50]  #creating a list
numbers.append(60)  #adds 60 at the end of the list
print numbers
numbers.insert(2, 25) #you can even add an element at a specific index
print numbers
```

```
Out: [10, 20, 30, 40, 50, 60]
Out: [10, 20, 25, 30, 40, 50, 60]
```

Notice how the list gets re-adjusted. All the elements after index 1 are moved 1 place right the new element is insert at index 2. you can delete an element from a list by using the keyword `del`.

```
numbers = [10,20,30,40,50]
del numbers[1] #this will remove element at index 1
print numbers
```

```
Out: [10, 30, 40, 50]
```

Notice how the list is re-adjusted. All the elements after index 1 are moved 1 place left to full the list. The length of list is also reduced by 1.

```
numbers = [10,20,30,40,50]  #creating a list
print numbers.pop() #removes and returns the last element in the list
print numbers
# More methods
print numbers.index(40) #returns the index of the element 40 if present
numbers = [30,20,40,10,50]
numbers.sort()  #sorts list
print numbers
```

```
Out: 50
Out: [10, 20, 30, 40]
Out: 3
Out: [10, 20, 30, 40, 50]
```

Strings as lists

In python, strings are lists. So, all the list operations can be done on strings.

```
name = "sreekanth"
print name[0]
print name[1:5] #slicing
print name + "kolamala" # concatination
#personal note: concatination does a shallow copy, use should use copy.deepcopy
a = range(10)
import copy
b = copy.deepcopy(a)
b.append(3)
print a
print b
```

```
Out: s
Out: reek
Out: sreekanthkolamala
Out: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Out: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 3]
```

Comprehensions

```
[expression(x) for x in some_list if condition]
```

expression(x) is the formula used for creating the elements. The elements of some_list is are used in expression(x). The `if condition` is also optional used for filtering. Some examples will solidify these concepts.

```
squares = [x**2 for x in range(10)]
print squares
#squares of even numbers:
squares = [x**2 for x in range(10) if x % 2 == 0]
print squares
squares = [x**2 for x in range(10) if not x % 2]
print squares
```

```
Out: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
Out: [0, 4, 16, 36, 64]
Out: [0, 4, 16, 36, 64]
```

Sets

Sets are like lists but duplicates are not allowed.

```
presentees = {1,5,6,7,10,1,6,6,5}
print presentees
presentees.add(10)
print presentees
```

```
Out: set([1, 10, 5, 6, 7])
Out: set([1, 10, 5, 6, 7])
```

Common mathematical set operations like union, intersection can be easily done.

```
setA = {1,2,3,4,5}
setB = {4,5,6,7,8}
print setA.union(setB)
print setA.intersection(setB)
```

```
Out: set([1, 2, 3, 4, 5, 6, 7, 8])
Out: set([4, 5])
```

Dictionaries

Dictionaries(commmonly shortened to dict) are {key,value} pair data structures. Each element in a dict is a pair of key and value. Keys have to be uquine. Keys can be thought as indices with names. Dicts are similar

to java maps.

```
wordFrequency = {'a':1, 'b':5, 'c':10, 'e':15}
#Here all the strings('a', 'b', 'c', 'e' are keys)
#all the numbers are values(1,5,10,15)
#each key has a associated value.
#For example, wordFrequency can be used to store frequency of alphabets in a string
print wordFrequency['a']
#key can be used like an index of a list.
del wordFrequency['a']
#deleting is similar to list.
print wordFrequency
print 'd' in wordFrequency  #find out if a key exists in dict using the 'in' keyword
```

```
Out: 1
Out: {'c': 10, 'b': 5, 'e': 15}
Out: False
```

Accessing the key that isn't present in the dict can result in an exception. This can be avoided by checking if that key is present before accessing it.

```
wordFrequency = {'a':1, 'b':5, 'c':10, 'e':15}
if('d' in wordFrequency):
  print wordFrequency['d']
else:
  print "not present"
```

```
Out: not present
```

Functions

Functions are small pieces of code that we can reuse by calling them instead of writing code multiple times. Their syntax:

```
def function_name(parameters):
  stmt
  stmt
  stmt
```

Few examples as usual will make things clear:

```
#function defination
def add(a,b):
  c = a + b
  print c
```

We define a function by using the keyword `def`. Notice that all the stmts after the first statement are indented. This is python's way to say that these statements are part of the function defination and not in the global context. Not indenting would throw an error. We can call this function like this:

```
#function calling
add(2,3)
add(2.3, 4.5)
add(a=2.3, b=4.5) #we can explicitly use the names of the parameter
                  #as defined in function defination
add(b=4.5, a=2.3) #we can also change the order of parameters.
```

```
Out: 5
Out: 6.8
Out: 6.8
Out: 6.8
```

Also notice that we were able to use the same add() function to add both ints and floats. Something we can't do in languages like C or Java.

Default parameters

We could give default values to function parameters. In which case, if the function is called without the parameter, the default value is used. If an actual parameter is given then the default one is ignored.

```
def print_series(n=10):
  for i in range(n):
    print i
print_series(5) #this will print till 5 #mental note: write this into 2 programs and eval, echo blocks
print_series()  #since no parameter is given, the default value 10 is assigned to n
                #and 0 till n are printed.
```

```
Out: 0
Out: 1
Out: 2
Out: 3
Out: 4
Out: 0
Out: 1
Out: 2
Out: 3
Out: 4
Out: 5
Out: 6
Out: 7
Out: 8
Out: 9
```

Fruitful Functions

Instead of printing, a function can also return a value.

```
#function that returns the square of its parameter
def square(number):
  return number**2
#function calling
a = square(4) #the value returned by the function can be assigned to a variable.
print a
```

```
Out: 16
```

Functions can return multiple values too.

```
def mean_mode_median(numbers):
  return 2,3,4

mean, mode, median = mean_mode_median(range(5))
print mean, mode, median
```

```
Out: 2 3 4
```

```
def square(number):
  return number**2
print map(square, range(10))
```

```
Out: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Instead of defining a function, we can use **anonymous function**. Anonymous function are like regular functions except they don't have a name. They defined using the keyword, `lambda`.

```
print map(lambda x: x**2, range(10))
```

```
Out: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Variable-length arguments

```
# program to take variable number of parameters
def max(*numbers):
  result = None
  for number in numbers:
    if number > result:
      result = number
  print result
#you can call max() with any number of parameters
max(30,20)
max(20,30,60,70,40,50)
max(20)
max()
```

```
Out: 30
Out: 70
Out: 20
Out: None
```

You can also mix normal parameters with variable number of parameters

```
def max(first, *numbers):
  result = first
  for number in numbers:
    if number > result:
      result = number
  print result
#you can call max() with atleast one parameter
max(30,20)
max(20,30,60,70,40,50)
max(20)
max() #this will throw an error
```

```
Out: 30
Out: 70
Out: 20
Out: Traceback (most recent call last):
Out:   File "<string>", line 11, in <module>
Out: TypeError: max() takes at least 1 argument (0 given)
```

Scope

Variables declared(appear for the first time in code) in the function(known as local variables) are not accessable outside the function.

```
def some_function():
  variable = 'this variable is available only inside the function some_function()'
  print variable
function()
print variable
```

```
Out: Traceback (most recent call last):
Out:   File "<string>", line 4, in <module>
Out: NameError: name 'function' is not defined
```

Vaariables declared outside the function(known as global variables) are available inside a function but can't be modified directly.

```
variable = 'global_variable'
def function():
    print variable

function()
```

```
Out: global_variable
```

If you try to modify, python will think that you are creating a new **local variable** that coincidentally has the same name as the global varaible.

```
variable = 'global_variable'
def function():
    # this age is different from the one declared in the global context
    variable = 'local_variable'
    print variable
print 'variable outside function before function call:', variable
function()
print 'age outside function after function call:', variable
```

```
Out: variable outside function before function call: global_variable
Out: local_variable
Out: age outside function after function call: global_variable
```

To use and modify a global variable, you need to declare it as global first. we do that by using the keyword **global**.

```
global_variable = 10
def function():
    global global_variable
    global_variable = 20  #this change will be visiable in the global variable

function()
print global_variable
```

```
Out: 20
```

Import statements

Often times, instead of writing code ourselves, we could you code written by others. This code is available in the form of libraries. We need to declare them before using them by writing an import statement. Its syntax:

```
import library_name
from library_name import class_or_function_name
```

Some examples: There is module called math that contains commonly used mathematical functions like square root{sqrt()} etc. To use them, we need to import the math module.

```
import math #you can call anything within math.
print math.sqrt(25) #the full name math.sqrt() is requeried.
print math.pi
print sqrt(25)
```

```
Out: 5.0
Out: 3.14159265359
Out: Traceback (most recent call last):
Out:   File "<string>", line 4, in <module>
Out: NameError: name 'sqrt' is not defined
```

```
from math import sqrt #only importing sqrt and not the whole of math module.
print sqrt(25)  #full name not required.
print pi  #can't use anything else.
```

```
Out: 5.0
Out: Traceback (most recent call last):
Out:   File "<string>", line 3, in <module>
Out: NameError: name 'pi' is not defined
```

We can also give alias(shortcut name) to the imported module. This way, we could use the shortcut instead of the full module name.

```
import math as m
print m.sqrt(25)
```

```
Out: 5.0
```

Some of these libraries are loaded along with python and hence need no import. Some examples:

Object Oriented Programming OOP in Python

Classes are neat way to organize code. Consider we write code for a box. A box has 3 properties-height, width, bredth. We can use variable like box_height, box_width, box_bredth to represent them. But doing this makes 3 seperate variables that seem to have no relation.

```
class Box(object):
  static_variable = 10
  def set_height(self, h):
    self.height = h

  def get_height(self):
    return self.height

  def __init__(self, h, w, b):
    self.height = h
```

```
    self.weight = w
    self.bredth = b

  def volume(self):
    return self.height*self.weight*self.bredth

  def __len__(self):
    return 10

box = Box(2,3,4)
print box.volume()
print len(box)
```

```
Out: 24
Out: 10
```

I know that's a lot of code. Don't worry. Before we We are creating a class called Box that has variable height

Magic methods

The beauty of python is in its magic methods. Lets us that we are defining a class to represent a family.

```
class Family(object):
  def __init__(self, mems):
    self.members = mems
family = Family(['ram', 'priya', 'ravi'])
print family.members
# We can use the len() functin to find out how many family members are there.
print len(family.members)
```

```
Out: ['ram', 'priya', 'ravi']
Out: 3
```

But with magic methods, we could simplify this.

```
class Family(object):
  def __init__(self, mems):
    self.members = mems
  def __len__(self):
    return len(self.members)
family = Family(['ram', 'priya', 'ravi'])
print len(family) #notice the change
```

```
Out: 3
```

What we have achivied here is: Family is type we build just now. The python language didn't know about this before now. len is built-in function. we defined our class to work seemlessly with python's built-in functions. **len(self)** is a magic method that gets called when len(family) is encountered.

```
class Family(object):
  def __init__(self, mems):
    self.members = mems
  def __repr__(self):
    return str(self.members)
  def __getitem__(self, name):
    return self.members[name]
```

```
family = Family(['ram', 'priya', 'ravi'])
print family
print family[1]
```

```
Out: ['ram', 'priya', 'ravi']
Out: priya
```

An excellent guide for magic methods(don't forget to check the table towards the end of the article): http://minhhh.github.io/posts/a-guide-to-pythons-magic-methods

self-note: discuss the keyword is

Inheritance

Let's create a special kind of family: south indian family

```
class South_Indian_Family(Family):
  pass

southy = South_Indian_Family(['ram', 'priya', 'ravi'])
print southy.members
print southy
```

```
Out: ['ram', 'priya', 'ravi']
Out: ['ram', 'priya', 'ravi']
```

```
#multiple inheritance
class A(object):
  pass

class B(object):
  pass

class C(A,B):
  pass
```

Overriding

Inheritance lets us use base class' methods in the derived class. That's very useful. However, sometimes, we might need a more specialized method instead of the default method.

```
class Animal(object):
  def make_sound(self):
    print 'animal sound'
  def eat(self):
    print 'animal eating'

class Dog(Animal):
  # this method is overriding the parent class' method
  def make_sound(self):
    print 'bow bow'
  def attack_visitor(self):
    print 'bak bak'
```

```
animal = Animal()
animal.make_sound()
animal.eat()
dog = Dog()
dog.eat() #the Animal class' eat() will be called, as Dog has not overridden this method.
dog.make_sound()  #the Dog class' make_sound() will be called.
dog.attack_visitor()
animal.attack_visitor() #error: remember attack_visitor() is not defined in Animal.
```

```
Out: animal sound
Out: animal eating
Out: animal eating
Out: bow bow
Out: bak bak
Out: Traceback (most recent call last):
Out:   File "<string>", line 21, in <module>
Out: AttributeError: 'Animal' object has no attribute 'attack_visitor'
```

Polymorphism

Lets code up a class Doctor that heals the animal when it is sick. To keep this simple, lets assume htat the doctor checks for the animal's bp(blood pressure) and the sound that the animal makes.

```
class Doctor(object):
  def heal(self, animal):
    print animal.bp
    animal.make_sound()
```

This Doctor works great. All animal lovers are going to this doctor for medical help. Few years later, a cat is

```
class Cat(Animal):
  def make_sound(self):
    print 'meow meow'
```

```
cat = Cat()
doctor = Doctor()
doctor.heal(cat)
```

```
Out: 100
Out: meow meow
```

Datahiding

Languages like C++, Java have public or private class variables. Python has none of them. All class variables are public. However, python programmers follow a naming convention to differentiate between public and private variables. **All variables that start with an underscore are deemmed as private and should not be touched by outside code**.

```
class Example(object):
  def __init__(self):
    self._private_variable = "Don't touch me! I'm private"
    self.public_variable = "I am public"
```

```
example = Example()
print example.public_variable
print example._private_variable #works, but never do this
```

```
Out: I am public
Out: Don't touch me! I'm private
```

Exception Handling

Let's write a program to for a simple calculator. Our calculator repeatedly asks 2 numbers and calculates their division.

```
while True:
    a = int(raw_input("enter numerator:"))
    b = int(raw_input("enter denominator:"))
    print a/b
```

```
Out: enter numerator:2.3
Out: enter denominator:4.5
Out: 0.511111111111
Out: enter numerator:5
Out: enter denominator:2
Out: 2
Out: enter numerator:
```

Looks good. This an never ending program(you can stop the program by pressing ctrl+z) lets see what happens when we give 0 for denominator

```
Out: enter numerator:4
Out: enter denominator:0
Out: Traceback (most recent call last):
Out:   File "<string>", line 4, in <module>
Out: ZeroDivisionError: integer division or modulo by zero
```

The program stops with an error message. ... we can avoid this by using a try-except block

```
while True:
    try:
        a = int(raw_input("enter numerator:"))
        b = int(raw_input("enter denominator:"))
        print a/b
    except ZeroDivisionError as e:
        print 'Denominator cannot be zero. Try again'
```

```
Out: enter numerator:5
Out: enter denominator:0
Out: denominator cannot be zero. Try again
Out: enter numerator:42
Out: enter denominator:5
Out: 8
Out: enter numerator:
```

Looks good. let's break this again. this time we will give invalid input.

```
Out: enter numerator:e
Out: Traceback (most recent call last):
Out:   File "<string>", line 3, in <module>
```

```
Out: ValueError: invalid literal for int() with base 10: 'e'
```

Giving a string instead of a number crashed the program. We can see from the error output, there was a ValueError error. Let's handle that.

```
while True:
    try:
        a = int(raw_input("enter numerator:"))
        b = int(raw_input("enter denominator:"))
        print a/b
    except ZeroDivisionError as e:
        print 'Denominator cannot be zero. Try again'
    except ValueError as e:
        print 'Please enter numbers'
```

```
Out: enter numerator:string
Out: please enter numbers
Out: enter numerator:3
Out: enter denominator:0
Out: denominator cannot be zero. Try again
Out: enter numerator:
```

We can also combine both exceptions into a single block like this:

```
  except (ZeroDivisionError, ValueError) as e:
    print 'Bad input. Please try again'
```

ZeroDivisionError and ValueError are two types of Exceptions. Infact there are derived from the class called Exception. A lot of exceptions are predefined by the language and all of them are derived from the class Exception. So, instead of using these, we can use the super class Exception.

```
while True:
    try:
        a = int(raw_input("enter numerator:"))
        b = int(raw_input("enter denominator:"))
        print a/b
    except Exception as e:
        print 'Bad input. Please try again'
```

However this is bad practice and should never be done. The reason being: this would mask exceptions which the programmer not intented to. Fyi, the above will run infinetely even if you press ctrl+z as that would also be caught as bad input.

Raising exceptions

```
class Queue(object):
    def __init__(self):
        self.elements = []

    def enqueue(self, elem):
        self.elements.append(elem)

    def deque(self):
        if not self.elements:
          raise Exception('Cannot deque: Empty queue')
        return self.elements.pop(0)
```

```
queue = Queue()
queue.enqueue(1)
print queue.deque()
print queue.deque()
```

```
Out: 1
Out: Traceback (most recent call last):
Out:   File "<string>", line 16, in <module>
Out:   File "<string>", line 10, in deque
Out: Exception: Cannot deque: Empty queue
```

Notice in the output, the message we coded was printed. We can also raise a more specific exception(like IndexError) instead of the generic superclass Exception. Or we could just create an exception. Let's create a exception called EmptyQueueError.

```
class EmptyQueueError(Exception): #notice instead of object, we are subclassing Exception
  pass

class Queue(object):
    def __init__(self):
        self.elements = []

    def enqueue(self, elem):
        self.elements.append(elem)

    def deque(self):
        if not self.elements:
          raise EmptyQueueError('cannot deque: empty queue')
        return self.elements.pop(0)

queue = Queue()
queue.enqueue(1)
print queue.deque()
print queue.deque()
```

```
Out: 1
Out: Traceback (most recent call last):
Out:   File "<string>", line 19, in <module>
Out:   File "<string>", line 13, in deque
Out: __main__.EmptyQueueError: cannot deque: empty queue
```

Notice instead of object, we are subclassing Exception instead of teh usaul object. This is because we the functionality of the Exception class.

Testing

Consider the following program to print whether a number is even or not

```
def even(num):
  if num % 2 == 0:
    print num + ' is even'
  else:
    print str(num) + ' is odd'
```

Now let's call with an odd number

```
even(3)
```

```
Out: 3 is odd
```

Now, let's call this function with an even number.

```
even(4)
```

```
Out: Traceback (most recent call last):
Out:   File "<string>", line 7, in <module>
Out:   File "<string>", line 3, in even
Out: TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

I purposefully placed an error on line 3 of the even() function. num is number and cannot be directly concatinated with a string. It should be converted into a string first. We did this on line 5. This is the reason that the program did not report error for the `even(3)`, because the program never hit line 3. The if condition failed and line 3 was never executed. Whereas `even(4)` was called, if condition passed and line 3 was hit. And then, the error came up.

In staticly typed langauages like C and java, this error would have been a compile time error. In python, it's a runtime error. Hence, we can't be sure that our code is error-free till we all the lines in our code are executed. This is where testing comes. No peice of software can be written in python without a through testing. Luckily, Testing in python is easy.

Let's say we want to test the function reverse_string().

```
def reverse_string(sentence):
    return ''.join(reversed(sentence))
```

What this function does is important but not how it does. This function reverses a string passed to it. We could have 3 test cases:

1) a random non-zero length string as input
2) a random non-zero length string with special symbols as input
3) an empty string

let's write the first test case and run it. We use the module unittest-a testing framework. We write test code seperate from our actual code. We create a class to hold all our test cases. you can name it everything but it should inherit from the TestCase class. Here, we name it `ReverseStringTestCase`.

```
from unittest import TestCase, main

class ReverseStringTestCase(TestCase):
    def test_reverse_string_random_string(self):
        self.assertEqual('random', reverse_string('modnar'))
```

Each test case is written as a method. `test_reverse_string_random_string()` will test the first test case. Remember our class inherited from the class TestCase. The cool thing about this class is it has few helper functions like `assertEqual(a,b)` that throws an error if a and b are not equal.

In our test case, we call `assertEqual('random', reverse_string('modnar'))`. If reverse_string() works correct, it should return 'random' which would be equal to the first argument of `assertEqual()`. Else, it throws an error and the test case will fail.

We run the test cases by calling `main()`. This is not a function we wrote, but is imported from unittest.

```
main()
```

```
Out: .
```

```
Out: ----------------------------------------------------------------------
Out: Ran 1 test in 0.000s
Out:
Out: OK
```

The output says 'ran 1 test' and gives the status 'OK'. This means all the test cases were successful. Our code has no bugs. Let's add the other two test cases as two methods and run them.

```
from unittest import TestCase, main

class ReverseStringTestCase(TestCase):
    def test_reverse_string_empty_string(self):
        self.assertEqual('', reverse_string(''))

    def test_reverse_string_random_string(self):
        self.assertEqual('random', reverse_string('modnar'))

    def test_reverse_string_random_string_not_equal(self):
        self.assertNotEqual('ranmod', reverse_string('modnar'))

main()
```

```
Out: ...
Out: ----------------------------------------------------------------------
Out: Ran 3 tests in 0.000s
Out:
Out: OK
```

We get an 'OK' status. All tests are passed. Now, Let's break things: Let's modify the reverse_string() method to give an incorrect result and see how the tests run.

```
def reverse_string(sentence):
    return ''
```

To run the tests we call `main()`:

```
main()
```

```
Out: .F.
Out: ======================================================================
Out: FAIL: test_reverse_string_random_string (__main__.ReverseStringTestCase)
Out: ----------------------------------------------------------------------
Out: Traceback (most recent call last):
Out:   File "<string>", line 10, in test_reverse_string_random_string
Out: AssertionError: 'random' != ''
Out:
Out: ----------------------------------------------------------------------
Out: Ran 3 tests in 0.000s
Out:
Out: FAILED (failures=1)
```

As expected, one of the test cases failed.

Lab programs

Exercise 1 Basics

a) Running instructions in Interactive interpreter and a Python Script

```
2 + 3
45 - 21.1
45/9 + 2*1.2
```

```
Out: 5
Out: 23.9
Out: 7.4
```

Write the above lines in file named script.py and that file with the following command.

```
run script.py
# output omitted as it would be the same as above
```

b) Write a program to purposefully raise Indentation Error and Correct it

```
#with error
print 42
  print 'some-string'
print True
```

```
Out:   File "<string>", line 3
Out:     print 'some-string'
Out:     ^
Out: IndentationError: unexpected indent
```

```
#with correction
print 42
print 'some-string'
print True
```

```
Out: 42
Out: some-string
Out: True
```

Exercise 2 Operations

a) Write a program to compute distance between two points taking input from the user (Pythagorean Theorem)

```
import math
x1 = int(raw_input("enter x1:"))
y1 = int(raw_input("enter y1:"))
x2 = int(raw_input("enter x2:"))
y2 = int(raw_input("enter y2:"))
distance = math.sqrt(math.pow(x2-x1, 2) + math.pow(y2-y1, 2))
print distance
```

b) Write a program add.py that takes 2 numbers as command line arguments and prints its sum.

Exercise 3 Control Flow

a) Write a Program for checking whether the given number is a even number or not.

```
number = int(raw_input("enter a number:"))
print "odd" if number % 2 else "even"
```

```
Out: enter a number:24
Out: even
```

b) Using a for loop, write a program that prints out the decimal equivalents of 1/2, 1/3, 1/4, . . . , 1/10

```
for i in range(2, 11):
    print 1.0/i
```

```
Out: 0.5
Out: 0.333333333333
Out: 0.25
Out: 0.2
Out: 0.166666666667
Out: 0.142857142857
Out: 0.125
Out: 0.111111111111
Out: 0.1
```

c) Write a program using a for loop that loops over a sequence. What is sequence ?

```
numbers = [2,3,4,5]
for i in numbers:
    print i
```

```
Out: 2
Out: 3
Out: 4
Out: 5
```

```
numbers = (2,3,4,5)
for i in numbers:
    print i
```

```
Out: 2
Out: 3
Out: 4
Out: 5
```

d) Write a program using a while loop that asks the user for a number, and prints a countdown from that number to zero.

```
number = int(raw_input("enter any number:"))
for i in range(number, -1, -1):
    print i
```

```
Out: enter any number:5
Out: 5
Out: 4
Out: 3
Out: 2
Out: 1
Out: 0
```

Alternate method

```
number = int(raw_input("enter any number:"))
for i in reversed(range(number + 1)):
    print i
```

Exercise 4 Control Flow - Continued

a) Find the sum of all the primes below two million.

```
def primes_till(limit):
    primes = {2}
    primes.update(set(range(3, limit, 2)))
    for i in range(3, limit/2):
        primes.difference_update(range(2*i, limit, i))
    return primes

print sum(primes_till(2000000))
```

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

b) By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

```
def fibonacci(limit):
    series, one, two = [], 1, 2
    while one <= limit:
        series.append(one)
        one, two = two, one + two
    return series

print sum(filter(lambda x: x % 2 == 0, fibonacci(4000000)))
```

Exercise 5 DS

a) Write a program to count the numbers of characters in the string and store them in a dictionary data structure

```
name = raw_input("enter a string:")
def countWordFrequency(name):
    wordFrequency = {}
    for char in name:
        if char in wordFrequency:
            wordFrequency[char] += 1
        else:
            wordFrequency[char] = 1
    return wordFrequency
print countWordFrequency(name)
```

```
Out: enter a string:sreekanth kolamala
Out: {'a': 4, ' ': 1, 'e': 2, 'h': 1, 'k': 2, 'm': 1, 'l': 2, 'o': 1, 'n': 1, 's': 1, 'r': 1, 't': 1}
```

Alternate solution

```
import collections
def countFrequency(name):
    wordFrequency = collections.defaultdict(int)
    for char in name:
        wordFrequency[char] += 1
    return dict(wordFrequency)
```

Alternate solution

```
import collections
name = raw_input("enter a string:")
print dict(collections.Counter(name))
```

```
Out: enter a string:sreekanth kolamala
Out: {'a': 4, ' ': 1, 'e': 2, 'h': 1, 'k': 2, 'm': 1, 'l': 2, 'o': 1, 'n': 1, 's': 1, 'r': 1, 't': 1}
```

b) Write a program to use split and join methods in the string and trace a birthday with a dictionary data structure.

I don't understand the question. Since can't give an answer. Sorry.

Exercise 7 Files

a) Write a program to print each line of a file in reverse order.

```
text_file = open('sample.txt', 'r')
fileContent = text_file.readlines()
for line in reversed(fileContent):
    print line
text_file.close()
```

```
Out: eight end
Out: four 5 $$ seven
Out:
Out: one two three
```

Alternate solution

```
with open('sample.txt', 'r') as text_file:
    fileContent = text_file.readlines()
    while fileContent:
        print fileContent.pop()
```

```
Out: eight end
Out: four 5 $$ seven
Out:
Out: one two three
```

b) Write a program to compute the number of characters, words and lines in a file.

```
with open('sample.txt', 'r') as text_file:
    lines = text_file.readlines()
    number_of_lines = len(lines)
    words = ''.join(lines).split()
    number_of_words = len(words)
```

```
    chars = ''.join(lines)
    number_of_chars = len(chars)
    print number_of_lines, number_of_words, number_of_chars
```

```
Out: 3 9 39
```

Exercise 8 Functions

a) Write a function ball_collide that takes two balls as parameters and computes if they are colliding. Your function should return a Boolean representing whether or not the balls are colliding. Hint: Represent a ball on a plane as a tuple of (x, y, r), r being the radius If (distance between two balls centers) <= (sum of their radii) then (they are colliding)

b) Find mean, median, mode for the given set of numbers in a list.

```
def mean(numbers):
    return float(sum(numbers)) / max(len(numbers), 1)

def median(numbers):
    if not numbers:
        return 0
    sortedNumbers = sorted(numbers)
    length = len(sortedNumbers)
    return (sortedNumbers[length/2] + sortedNumbers[(length-1)/2])/2.0

#if you are going to sort numbers, there is simpler way to find mode
def mode(numbers):
    frequencies = countWordFrequency(numbers)
    maximum = None
    result = None
    for number, frequency in frequencies.items():
        if maximum < frequency:
            maximum = frequency
            result = number
    return result

numbers = [2,3,5,2,10,25]
print mean(numbers), median(numbers), mode(numbers)
```

Exercise 9 Functions - Continued

a) Write a function nearly_equal to test whether two strings are nearly equal. Two strings a and b are nearly equal when a can be generated by a single mutation on b.

```
def nearlyEqual(a,b):
    if len(a) != len(b):
        return False
    difference = False
    for chara, charb in zip(a, b):
        if chara is not charb:
            if difference:
                return False
            else:
```

```
                difference = True
    return True

print nearlyEqual("sree", "srek")
print nearlyEqual("sree", "sek")
print nearlyEqual("sree", "serk")
```

```
Out: True
Out: False
Out: False
```

b) Write a function dups to find all duplicates in the list.

```
def dups(array):
    import collections
    frequencies = collections.Counter(array)
    return [key for key, value in frequencies.items() if value > 1]

print dups([2,3,1,2,5,8,9,8])
```

```
Out: [2, 8]
```

c) Write a function unique to find all the unique elements of a list.

```
def unique(array):
    import collections
    frequencies = collections.Counter(array)
    return [key for key, value in frequencies.items() if value == 1]

print unique([2,3,1,2,5,8,9,8])
```

```
Out: [1, 3, 5, 9]
```

Exercise 10 Functions - Problem Solving

a) Write a function cumulative_product to compute cumulative product of a list of numbers.

```
def cumulative_product(numbers):
    result = 1
    for number in numbers:
        result *= number
    return result

print cumulative_product(range(1,5))
```

```
Out: 24
```

Alternate solution

```
def cumulative_product(numbers):
    import operator
    return reduce(operator.mul, numbers, 1)

print cumulative_product(range(1,5))
```

```
Out: 24
```

b) Write a function reverse to reverse a list. Without using the reverse function.

```
def reverse(elements):
    result = []
    length = len(elements)
    for i in range(length):
        result.append(elements[length-1-i])
    return result

print reverse([1,2,3,4,5])
```

```
Out: [5, 4, 3, 2, 1]
```

Alternate solution

```
def reverse(elements):
    length = len(elements)
    return [elements[length-1-i] for i in range(length)]

print reverse([1,2,3,4,5])
```

```
Out: [5, 4, 3, 2, 1]
```

c) Write function to compute gcd, lcm of two numbers. Each function shouldn't exceed one line.

```
def gcd(a,b):
    return [x for x in range(1, max(a, b)) if a % x == 0 and b % x == 0].pop()

def lcm(a,b):
    return [x for x in range(max(a, b), a*b+1) if x % a == 0 and x % b == 0].pop(0)

print gcd(5, 7)
print gcd(5, 25)
print lcm(20, 30)
print lcm(3, 5)
```

```
Out: 1
Out: 5
Out: 60
Out: 15
```

Exercise 11 Multi-D Lists

a) Write a program that defines a matrix and prints

```
matA = [[1,2,3],
        [4,5,6],
        [7,8,9]]

matB = [[3,2,1],
        [6,5,4],
        [9,8,7]]
```

```
print matA
print matB
```

```
Out: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Out: [[3, 2, 1], [6, 5, 4], [9, 8, 7]]
```

b) Write a program to perform addition of two square matrices

```
def addMatrix(a,b):
    import operator
    return [map(operator.add, rowA, rowB) for rowA, rowB in zip(a, b)]

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

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

print addMatrix(matA, matB)
```

```
Out: [[4, 4, 4], [10, 10, 10], [16, 16, 16]]
```

c) Write a program to perform multiplication of two square matrices

```
def multiply_matrix(mat_a, mat_b):
    a = len(mat_a)
    b = len(mat_b[0])
    result = [[0 for _ in range(b)] for _ in range(a)]
    for i in range(a):
        for j in range(b):
            for k in range(len(mat_a[0])):
                result[i][j] += mat_a[i][k]*mat_b[k][j]
    return result

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

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

print multiply_matrix(matA, matB)
```

```
Out: [[42, 36, 30], [96, 81, 66], [150, 126, 102]]
```

Alternate solution

```
def tranpose(a):
    no_of_rows = len(a)
    no_of_columns = len(a[0])
    return [[a[i][j] for i in range(no_of_rows)] for j in range(no_of_columns)]
```

```
def multiply_matrix(mat_a, mat_b):
    import operator
    trans_b = tranpose(mat_b)
    return [[sum(map(operator.mul, mat_a[i], trans_b[j]))
             for j in range(len(trans_b))] for i in range(len(mat_a))]

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

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

print multiply_matrix(matA, matB)
```

```
Out: [[42, 36, 30], [96, 81, 66], [150, 126, 102]]
```

Exercise - 14 GUI, Graphics 1. Write a GUI for an Expression Calculator using tk 2. Write a program to implement the following figures using turtle

Exercise 15 Testing

a) Write a test-case to check the function even-numbers which return True on passing a list of all even numbers

```
from unittest import TestCase, main

def even_numbers(numbers):
    numbers = map(lambda x: x % 2 == 0, numbers)
    return all(numbers)

class EvenTestCase(TestCase):
    def test_even_numbers_empty_list(self):
        self.assertEqual(True, even_numbers([]))

    def test_even_numbers_all_even_numbers(self):
        self.assertTrue(even_numbers(range(0,25,2)))

    def test_even_numbers_all_one_odd_number(self):
        self.assertFalse(even_numbers([2,4,7,2,10]))

main()
```

```
Out: ...
Out: ----------------------------------------------------------------------
Out: Ran 3 tests in 0.000s
Out:
Out: OK
```

Ideally, actual code and test code shouldn't be written in the same file. The test code should be in a separate file. We will use this approach in the next exercise.

b) Write a test-case to check the function reverse_sting which returns the reversed string

```
#write this code in a file named rever.py
def reverse_string(sentence):
    return ''.join(reversed(sentence))
```

```
#write this code in a file named test_rever.py
#and then run this file on the python interpreter: run test_rever.py
from unittest import TestCase, main
from rever import reverse_string

class ReverseStringTestCase(TestCase):
    def test_reverse_string_empty_string(self):
        self.assertEqual('', reverse_string(''))

    def test_reverse_string_random_string(self):
        self.assertEqual('random', reverse_string('modnar'))

    def test_reverse_string_random_string_not_equal(self):
        self.assertNotEqual('ranmod', reverse_string('modnar'))

if __name__ == '__main__':
    main()
```

```
Out: ...
Out: ----------------------------------------------------------------------
Out: Ran 3 tests in 0.000s
Out:
Out: OK
```

Exercise 16 Advanced

a) Build any one classical data

```
class Queue(object):
    def __init__(self):
        self.elements = []

    def enqueue(self, elem):
        self.elements.append(elem)

    def deque(self):
        return self.elements.pop(0)

    def __len__(self):
        return len(self.elements)

    def __repr__(self):
        return str(self.elements)

queue = Queue()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
queue.enqueue(4)
print 'elements in the queue are' , queue
```

```
print queue.deque()
print queue
print 'queue has', len(queue), 'elements'
```

```
Out: elements in the queue are [1, 2, 3, 4]
Out: 1
Out: [2, 3, 4]
Out: queue has 3 elements
```

b) Write a program to solve knapsack

```
def knapsack(items, max_weight):
    best_values = [[0] * (max_weight + 1)
                   for _ in range(len(items) + 1)]

    for i, (value, weight) in enumerate(items):
        for capacity in range(max_weight + 1):
            if weight > capacity:
                best_values[i+1][capacity] = best_values[i][capacity]
            else:
                candidate1 = best_values[i][capacity]
                candidate2 = best_values[i][capacity - weight] + value
                best_values[i+1][capacity] = max(candidate1, candidate2)

    reconstruction = []
    j = max_weight
    for i in range(len(items), 0, -1):
        if best_values[i][j] != best_values[i - 1][j]:
            reconstruction.append(items[i - 1])
            j -= items[i - 1][1]

    reconstruction.reverse()

    return best_values[len(items)][max_weight], reconstruction

items = [(1,3), (3,5), (5,4), (8,3)]
best_value, best_items = knapsack(items, 8)
print 'best value is', best_value
print 'best items are', best_items
```

```
Out: best value is 13
Out: best items are [(5, 4), (8, 3)]
```

My text!

Appendix

https://powerfulpython.com/store/p/restful-api-server/