Ex 1

s1 = [20, 30, 40]

s2 = [2, 5]

final\_ans = 0

total = "und"

prompt1 = " 20, 30, and 40 all / by 2 and 5?"

prompt2 = " 20, 32, and 40 all / by 2 and 5?"

for i in s1:

x = i

for i in s2:

y = i

if (x % y == 0):

final\_ans = final\_ans + 1

else:

final\_ans = final\_ans

if final\_ans == 6:

total = "True"

else:

total = "False"

print(first\_prompt)

print(total)

Ex 2

def sqroot\_newton(a):

x = a/2

y = (x+(a/x))/2

while x != y:

x = y

y = (x+(a/x))/2

return x

print sqroot\_newton(17.0)

Ex 3

import cImage as image

img = image.Image("hat.gif")

newimg = image.EmptyImage(img.getWidth(),img.getHeight())

win = image.ImageWin()

for row in range(1,img.getHeight()-1):

for col in range(1,img.getWidth()-1):

p = img.getPixel(col,row)

top = img.getPixel(col,row-1)

right = img.getPixel(col+1,row)

bottom = img.getPixel(col,row+1)

left = img.getPixel(col-1,row)

newred = (top.getRed()+right.getRed()+bottom.getRed()+left.getRed())/4

newpixel = image.Pixel(newred,newgreen,newblue)

newimg.setPixel(col,row,newpixel)

newimg.draw(win)

win.exitonclick()

Ex 4

a = ["unc", "duke", "ncstate"]

b = ["ncstate", "duke", "unc"]

c = ["unc", "duke", "vt"]

d = ["unc", "unc", "unc"]

a.sort()

b.sort()

c.sort()

d.sort()

def has\_same\_elements(x,y):

return x == y

print has\_same\_elements(a,b) # True

print has\_same\_elements(a,c) # False

print has\_same\_elements(a,d) # False

Ex 5 – email

import re

faculty = open("faculty.html","r")

emails = re.findall(r'[\w\.-]+@[\w\.-]+', faculty.read())

emails = list(set(emails))

print emails

Ex 6

import string

fp = open("courses2.txt", "r")

hist = dict()

for line in fp:

line = line.strip()

row = line.split()

courses = row[0]

instructors = row[1]

if courses in hist:

hist[courses].append(instructors)

else:

hist[courses] = [instructors]

print hist

course\_numbers = list(hist.keys())

print course\_numbers

print hist.get('523')

course\_list = []

for key in hist:

if 'Capra' in hist[key]:

course\_list.append(key)

print course\_list

instructor\_names = list(hist.values())

instructor\_list = []

for i in instructor\_names:

for name in i:

if name not in instructor\_list:

instructor\_list.append(name)

print instructor\_list

Ex 6 output

{'760': ['Capra'], '512': ['Haas'], '884': ['Kelly'], '523': \

['Capra', 'Haas', 'Mostafa'], '509': ['Arguello', 'Kelly', 'Losee']}

['760', '512', '884', '523', '509']

['Capra', 'Haas', 'Mostafa']

['760', '523']

['Capra', 'Haas', 'Kelly', 'Mostafa', 'Arguello', 'Losee']

Ex 7 elevator

class Elevator:

def \_\_init\_\_(self):

self.floor = 1

self.num\_pass = 0

self.door\_open = False

def get\_floor(self):

return self.floor

def get\_num\_pass(self):

return self.num\_pass

def get\_door(self):

return self.door\_open

def \_\_str\_\_(self):

return "floor=" + str(self.floor) + ", passengers=" \

+ str(self.num\_pass) + ", door open=" + str(self.door\_open)

def call\_to\_floor(self, newfloor):

self.door\_open = False

self.floor = newfloor

def enter\_pass(self, num):

self.door\_open = True

self.num\_pass = self.num\_pass + num

def exit\_pass(self, num):

self.door\_open = True

self.num\_pass = self.num\_pass - num

e = Elevator()

print e

e.call\_to\_floor(9)

print e

e.enter\_pass(3)

print e

e.call\_to\_floor(4)

print e

e.exit\_pass(1)

print e

Ex 7 output

floor=1, passengers=0, door open=False

floor=9, passengers=0, door open=False

floor=9, passengers=3, door open=True

floor=4, passengers=3, door open=False

floor=4, passengers=2, door open=True

Ex 8 – Cards

import random

class Card(object):

suit\_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']

rank\_names = [None, 'Ace', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King']

def \_\_init\_\_(self, suit=0, rank=2):

self.suit = suit

self.rank = rank

def \_\_str\_\_(self):

return '%s of %s' % (Card.rank\_names[self.rank], Card.suit\_names[self.suit])

def \_\_cmp\_\_(self, other):

# check the ranks

if self.rank > other.rank: return 1

if self.rank < other.rank: return -1

# ranks are the same, so check the suits

if self.suit > other.suit: return 1

if self.suit < other.suit: return -1

# suits and ranks are the same, so tie

return 0

class Deck(object):

def \_\_init\_\_(self):

self.cards = []

for suit in range(4):

for rank in range(1,14):

card = Card(suit, rank)

self.cards.append(card)

def \_\_str\_\_(self):

res = []

for card in self.cards:

res.append(str(card))

return '\n'.join(res)

def pop\_card(self):

return self.cards.pop()

def add\_card(self, card):

self.cards.append(card)

def shuffle(self):

random.shuffle(self.cards)

def move\_cards(self, hand, num):

for i in range(num):

hand.add\_card(self.pop\_card())

class Hand(Deck):

''' Hand inherits from Deck. '''

def \_\_init\_\_(self,label=''):

self.cards = []

self.label = label

def has\_pair(self):

res = []

result = False

for card in self.cards:

res.append(card.rank)

for i in res:

res.count(i)

if res.count(i) >= 2:

result = True

return result

def has\_twopair(self):

res = []

result = False

for card in self.cards:

res.append(card.rank)

res2 = []

for i in res:

res2.append(res.count(i))

n = 0

for second\_i in res2:

if second\_i == 2:

n = n+1

if n >= 4:

result = True

return result

def has\_flush(self, num):

res = []

result = False

for card in self.cards:

res.append(card.suit)

for i in res:

res.count(i)

if res.count(i) == num:

result = True

return result

def classify(self):

label = ''

if h.has\_flush(7):

label = "flush"

return label

if h.has\_twopair():

label = "two pair"

return label

if h.has\_pair():

label = "pair"

return label

else:

label = "high card"

return label

d = Deck()

h = Hand()

d.shuffle()

#allows for running 5 or 7 card hands

d.move\_cards(h,7)

print "Your hand is:"

print h

print "You have a pair:"

print h.has\_pair()

print "You have two pair:"

print h.has\_twopair()

print "You have a flush:"

print h.has\_flush(7)

print "Your hand is:"

print h.classify()

Ex 8 output

Your hand is:

3 of Clubs

x5 or 7

You have a pair: True

You have two pair: True

You have a flush: False

Your hand is: two pair

Emma.py

import string

def process\_file(filename):

hist = dict()

fp = open(filename)

for line in fp:

process\_line(line,hist)

return hist

def process\_line(line,hist):

line = line.replace('-',' ')

for word in line.split():

word = word.strip(string.punctuation + string.whitespace)

word = word.lower()

hist[word] = hist.get(word,0) +1

def total\_words(hist):

return sum(hist.values())

def different\_words(hist):

return len(hist)

def most\_common(hist):

t = []

#Nice to return a list of tuples becuase it allows you to loop through and do stuff with the key and the value

for key, value in hist.items():

t.append((value, key))

t.sort(reverse = True)

return t

def subtract (d1, d2):

result = dict()

for key in d1:

if key not in d2:

result[key] = None

return result

hist = process\_file('emma\_ch1.txt')

t = most\_common(hist)

words = process\_file('words.txt')

Lists

a = [81,82,83]

b = [81,82,83]

print(a is b) #False

print(a == b) #True

Recursion

def fac(n):

if n == 0:

return 1

else:

tmp = n \* fac(n-1)

return tmp

print fac(4)

#explanation of recursion

#fac(3)

# tmp = 3 \* fac(2)

# tmp = 2 \* fac(1)

# tmp = 1 \* fac(1)

# 1

# 2

# 6

def fib(n):

#print "time"

if n == 0:

return 0

elif n == 1:

return 1

else:

return fib(n-1) + fib (n-2)

print fib(4)

t = [1, 3, 5, 7, 9]

def listsum(t):

sum = 0

for i in t:

sum = i + sum

return sum

print listsum(t)

def listsum2(t):

if len(t) == 1:

return t[0]

else:

tmp = t[0] + listsum2(t[1:])

return tmp

print listsum2(t)

print "Example 5:"

word = "python"

def str\_reverse(word):

#backwards = ""

if len(word) == 1:

return word

else:

backwards = word[-1] + str\_reverse(word[:-1])

return backwards

print str\_reverse(word)

Output Recursion

24

3

25

25

Example 5:

Nohtyp

######################

Data Types

• Integer: 4, 123

• Float: 3.14, 89.75

• String: “Hello”, ‘Hello’, ‘3.75’

• **type()** will return the type of an object

var names must start w/ \_ or letter

• A statement is an instruction to be executed

• An expression combines values, variables, operators and functions and are evaluated to produce a value

Boolean evals to:

True or False

==

logical operators: and, or, not

For loop – range()

• range(n) – returns a list [0 .. n-1]

• range(n,m) – returns a list [n .. m-1]

**Debugging**

• Avoid (major) debugging by:  Start small

 Keep it working / small victories  Example from INTPY

• Hints

 Test boundary conditions

 Know your error messages

• 90% = ParseError, TypeError, NameError, ValueError • Examples from INTPYDebugging Error Types

• ParseError – syntax error

 Ex: missing parens, quotes, commas

 Try: comment out line, see what errors change  Try: narrow the source of the error

• TypeError – incompatible objects Ex: try to add an int and str

 Often math/expression statements Try: printvalues

**Debugging Error Types**

• NameError – use a var before it has a value

 Often caused by typos, speeling mistaches, mis- remembering var/function name

 Try: use search feature of editor

• ValueError – pass wrong type parameter to a function

Compound statements:

A **header** line that ends with a colon

A **body** that is indented 4 spaces and has one or more other Python statements

Function takes param, arg passed in

docstring

def greet (name):

'''Print a greeting to name.''' print "Hello,", name

In Python shell:

>>> greet.\_\_doc\_\_

**Global vars**

def square(x):

y = num \* num

return y

• First, Python looks for a variable in the local scope of the function.

• If it finds it there, it will use that one.

• If not, then it will look in the global scope.

num = 4

answer = square(num) print answer

**Accumulator pattern**

X = x+1

More Iteration – While loop

def sumTo(aBound):

""" Return the sum of 1+2+3 ... n """ theSum = 0

aNumber = 1

while aNumber <= aBound:

theSum = theSum + aNumber

aNumber = aNumber + 1 return theSum

print(sumTo(4)) print(sumTo(1000))

While vs. For

• Use for if you know the number of times you need to iterate

 Traversing a list of elements

 Do something 10 times (e.g. can use range)  Definite iteration

• Use while if you need to iterate until some condition is met

 Indefinite iteration

break can be used to exit a loop

def find\_brad(namelist): for name in namelist:

print name

if name == "Brad":

print "Found Brad!"

break;

find\_brad(["Amy", "Brad", "Cathy"])’ Break with an infinite loop

def type\_hello():

while True:

line = raw\_input ("Please type hello: ")

if line == "hello":

break

type\_hello();

Strings • Concatenation operator: +

fruit = "banana"

bakedGood = " nut bread"

print(fruit + bakedGood)

String Length

• Watch out!

fruit = "Banana"

sz = len(fruit)

# Which of the next two lines?

#lastch = fruit[sz] #WRONG

lastch = fruit[sz-1]

print(lastch)

**Strings immutable**

Can’t use [] to get letter

String Traversal using While

s = "UNC Tar Heels"

i=0

while i < len(s):

print s[i]

i=i+ 1

List Deletion

accc = ["duke", "gatech", "miami", "unc",

"pitt", "uva", "vatech"]

del accc[0]

We can clone a list

a = [81, 82, 83]

b = a[:] # clone by slice

print (a == b)

print(a is b)

a[0] = 71 print b

b[-1] = 73 print a print b

**Pure functions** as opposed to mutator don’t have side effects

Reduce

• Combine a sequence of elements into a single value def add\_all(t):

result = 0

for s in t:

result += s

return result

a = [10, 20, 80, 90]

print add\_all(a)

Tuples

• Tuples are similar to lists, but are **immutable**

t = 'a', 'b', 'c' #ok

t = ('a', 'b', 'c') #typical

Tuple operators

• Brackets and slices work similar to lists

• But tuples are immutable

t = ('a', 'b', 'c', 'd', 'e') print t[2]

print t[1:3]

t[2] = 'z' #error

Tuple assignment

• Use tuples to swap variables without a temp var temp = a

a= b

b = temp

a, b = b, a #tuple assignment

addr = 'santa@northpole.org'

uname, domain = addr.split('@')

Tuple as return values

• Functions can use tuples to return multiple values

t = divmod(7, 3)

print t

print type(t)

quot, rem = divmod(7, 3)

print quot

print rem

def min\_max(t):

return min(t), max(t)

print min\_max([1, 3, 5, 7])

(RUN)

(2, 1)

<type 'tuple'>

2

1

(1, 7)

Variable-length arguments

• Prefix param name with a \* to gather args into a tuple

• Gather&scatter

def printall(\*args): #gather

print args

printall(1, 2, 3)

t = (7, 3)

print divmod(t) #error

print divmod(\*t) #scatter

(RUN)

(1, 2, 3)

(2, 1)

Lists of tuples & zip

• Zip takes 2 or more sequences and “zips” them into a list of tuples

s = 'abc' #string

t = [0, 1, 2] #list

z = zip(s, t)

print z

for aletter, anumber in z: #tuple assignment

print aletter, anumber

(RUN)

[('a', 0), ('b', 1), ('c', 2)]

a 0

b 1

c 2

REGEX

Import re

match = re.search(r'pi+', 'piiig')

print match.group()

match = re.search(r'i+', 'piigiiii')

print match.group()

match = re.search(r'\d\s\*\d\s\*\d', 'xx1 2 3xx')

print match.group()

match = re.search(r'\d\s\*\d\s\*\d', 'xx12 3xx')

print match.group()

match = re.search(r'\d\s\*\d\s\*\d', 'xx123xx')

print match.group()

(RUN)

piii

ii

1 2 3

12 3

123

Matching Alternatives

• Square brackets – any listed char can match

[ab] means either a or b can match

[a-d] matchesaorborcord

• Use caret for negation

[^a-d] matches any char except a, b, c, or d

Group Extraction

• Parenthesis can be used to group parts

• These parts are then available in the group()

• Groups are referenced 1, 2, 3... left to right

import re

a = "A765-2781-ZFQ"

match = re.search(r'([AB])([0-9]+)-([0-9]+)-([A-Z0-9]+)',a)

print match.group()

print match.group(1)

print match.group(2)

print match.group(3)

print match.group(4)

(RUN)

A765-2781-ZFQ

A

765

2781

ZFQ

Alternatives w/more than one char

a = "crate"

b = "state"

match = re.search(r'(cr|st)ate',a)

print match.group()

match = re.search(r'(cr|st)ate',b)

print match.group()

#prints out crate and state

##############

Dictionaries

• So far we have seen *sequential* collections

Strings, lists, tuples

Have an order from left to right

Use integer indices to access values

• Dictionaries are a *mapping* type

Unordered, associative collection

Mapping from *keys* to *values*

• Keys can be any immutable type

• Values can be any Python data object

(including other collections)

Dictionaries are mutable

Dictionary Example

e2s = {}

e2s['one'] = 'uno' e2s['two'] = 'dos' e2s['three'] = 'tres' print e2s

print e2s['two']

**Output:**

**Ordering is undefined**

{'three': 'tres', 'two': 'dos', 'one': 'uno'}

dos

Dictionary Operations

inv = {'apples': 430, 'bananas':312,

'oranges': 523, 'pears':217}

print inv

inv['pears'] = 0

inv['bananas'] += 200

del inv['oranges']

print inv

print len(inv)

**Output:**

**Dictionaries are mutable**

{'pears': 217, 'apples': 430, 'oranges': 523,

'bananas': 312}

{'pears': 0, 'apples': 430, 'bananas': 512}

Dict Ops

inv = {'apples': 430, 'bananas':312, 'oranges': 523, 'pears':217}

for akey in inv.keys():

print "The key", akey, "maps to value", inv[akey]

tmp = list(inv.keys())

print tmp

for akey in inv:

print akey, inv[akey]

(RUN)

The key bananas maps to value 312

['pears', 'apples', 'oranges', 'bananas']

pears 217]

bananas 312

--more—

for (k,v) in inv.items():

print k,v

for k in inv:

print k, inv[k]

#both print key, val on same like above

\*WILL get error if try to reference a key in dict that doesn’t exist – can manually set it

#print inv['kiwi'] # error!

print inv.get('kiwi',0)

RECURSION

Base Case

• The base case is what allows the algorithm to stop recursing

• It should represent a case that is *trivial*

Meaning that it cannot be decomposed further

and that the solution (to the base case) is simple

• Examples:

A list of length 1 is always is sorted order

The sum of elements in a list of length 1 is just the value of the one element.

Change state to move toward base case

• The algorithm should, for each recursive call, move closer to the base case

• Examples

A list shrinks by one

A number is divided by some factor

Function should call itself, recursively

• If the function is not in the base case

It will typically perform some operation

And call itself with an argument that moves things closer to the base case.

###############

Valid XML Documents

• XML documents are “stricter” than HTML

Balanced tags/closing tags required

Case-sensitive

Proper nesting

• DTDs and XML Schemas

XML documents must conform to a DTD or XML Schema

XML Examples

http://www.w3schools.com/svg/svg\_example.asp

• SVG

http://www.w3schools.com/svg/svg\_inhtml.asp

• RDF

http://en.wikipedia.org/wiki/Resource\_Description\_Framework

• RSS

http://en.wikipedia.org/wiki/RSS\_%28file\_format%29

• VoiceXML

http://www.w3.org/TR/voicexml20/#dml2.2

XML Parsing in Python

• Python library: xml.etree.ElementTree

• ElementTree represents the whole XML document

• Element represents a single node in the tree

<?xml version="1.0"?>

<data>

<country name="Liechtenstein">

<rank>1</rank>

<year>2008</year>

<gdppc>141100</gdppc>

<neighbor name="Austria" direction="E"/>

<neighbor name="Switzerland" direction="W"/>

</country>

</data>

import xml.etree.ElementTree as ET

tree = ET.parse('country\_data.xml')

root = tree.getroot()

print root

print root.tag

print root.attrib

print root[0][1].text

for child in root:

print child.attrib['name']

for n in root.iter('neighbor'):

print n.attrib

for n in root[2].iter('neighbor'):

print n.attrib

(RUN)

<Element 'data' at 0x1004c4450>

data

{}

2008

Liechtenstein

{'direction': 'E', 'name': 'Austria'}

import xml.etree.ElementTree as ET

tree = ET.parse('country\_data.xml')

root = tree.getroot()

for country in root.findall('country'):

rank = country.find('rank').text

name = country.get('name')

print name, rank

**.findall** finds only elements with the specified tag that are direct children of the current element

**.find** will find only the first match

**.text** accesses the element’s text

**.get** accesses the element’s attributes

OBJECTS

Object-Oriented Programming

• OOP is a different way of thinking about writing programs

• Procedural:

cook(oven)

I have a cook() procedure that I pass oven to

• Object-oriented:

oven.cook()

I have an oven object that I ask to invoke its cook method

Object-Oriented Programming • Advantages

Provides an organization for our code, functions, methods

This organization can mirror real-world objects

Is often a logical way to think about our software

architecture

Objects can be self-contained – methods you need are part of the object

Objects

• In Python, every value is an object

Integers, lists, dict, turtles – are all objects

• Programs manipulate objects in two ways

Perform some computation with the object

Ask the object to perform one of its methods

Objects have:

State – what the object knows about itself

• E.g., position, color, capacity

Methods – actions that it can perform

• E.g., move, rotate, simplify, length

* CLASS represents the self.x etc.

Initializer method

(also called the constructor)

* Called every time we create a new instance of Point

STRINGS

\_\_str\_\_ is a special method

Python will call this method on an object when you try to print it.

Previously, Python was using a default \_\_str\_\_ method for the object.

We **override** this method by defining our own \_\_str\_\_.

INHERITANCE

\_\_cmp\_\_

• Allows use of <, >, == operators with objects • Rules:

Take two objects

Return positive number if first is greater

Return negative number if second is greater Return zero if both are equal

def \_\_cmp\_\_(self, other):

# check the ranks

if self.rank > other.rank: return 1

if self.rank < other.rank: return -1

# ranks are the same, so check the suits if self.suit > other.suit: return 1

if self.suit < other.suit: return -1

# suits and ranks are the same, so tie return 0

“veneer” (or “thin”) methods – like pop\_card in Dec class

Inheritance

• Inheritance allows us to define a new class that “inherits” methods and attributes from an existing object.

• We can then modify the new object.

• Example:

Hands of cards are similar to decks of cards But have some important differences

class Hand(Deck):

''' Hand inherits from Deck. '''

Inheritance Pros and Cons

• Pros

Can reduce amount of code / encourage code reuse

Sometimes reflects the real-world structure of objects

• Cons

Can make programs harder to read, understand,debug

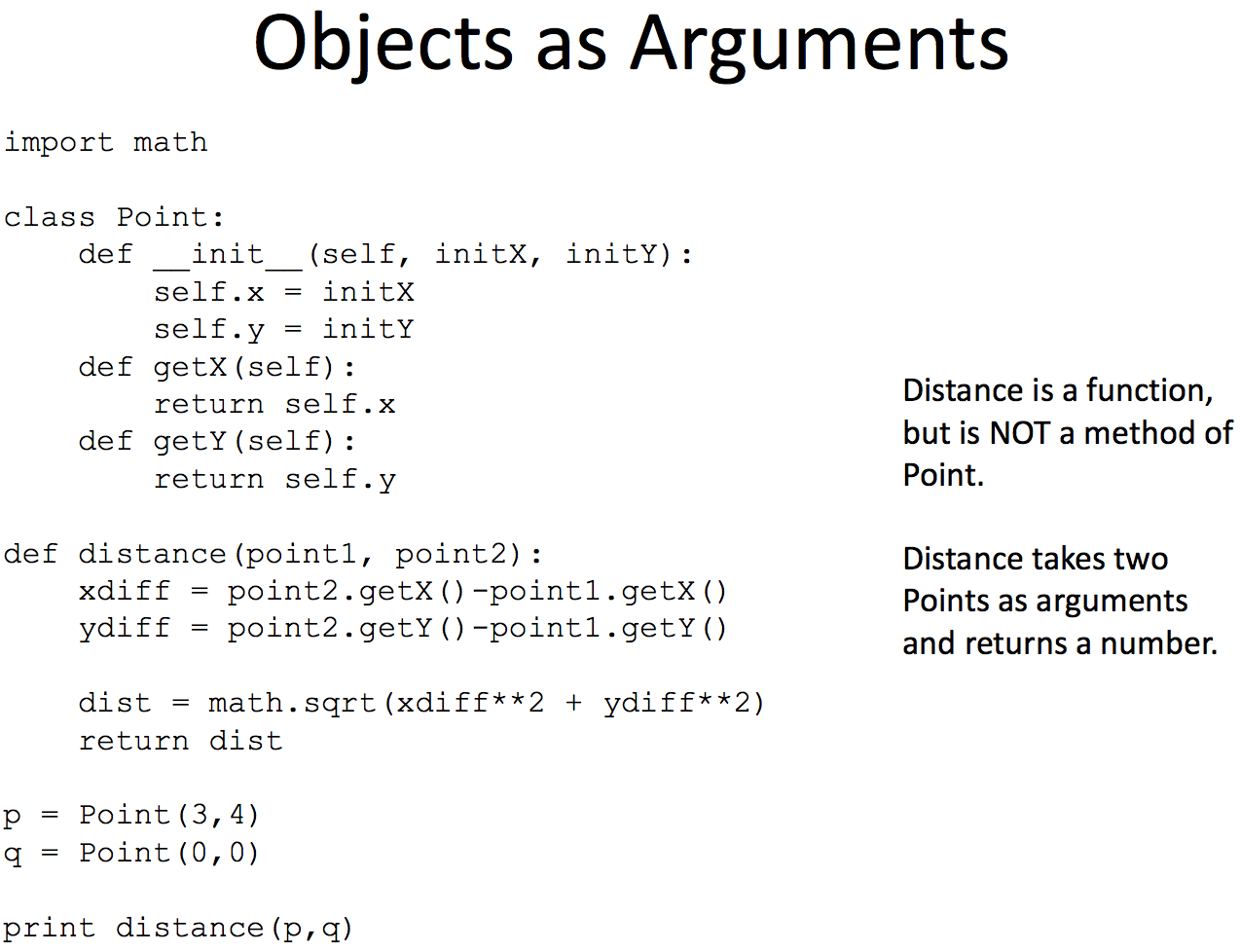
Code is located in different places/classes

Often inheritance is not needed... there are other ways to structure things

**Deterministic -** A process that is repeatable and predictable.

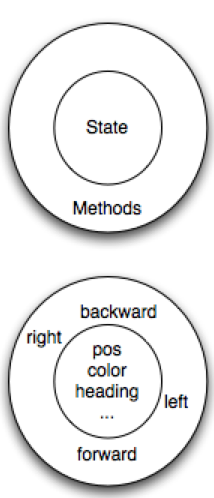
**Argument -** A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function. The argument can be the result of an expression which may involve operators, operands and calls to other fruitful functions.

**Body -** The second part of a compound statement. The body consists of a sequence of statements all indented the same amount from the beginning of the header. The standard amount of indentation used within the Python community is 4 spaces.

**compound statement**

header - which begins with a keyword determining the statement type, and ends with a colon.

body - containing one or more statements indented the same amount from the header.

**docstring -** If the first thing in a function body is a string (or, we’ll see later, in other situations too) that is attached to the function as its \_\_doc\_\_ attribute.

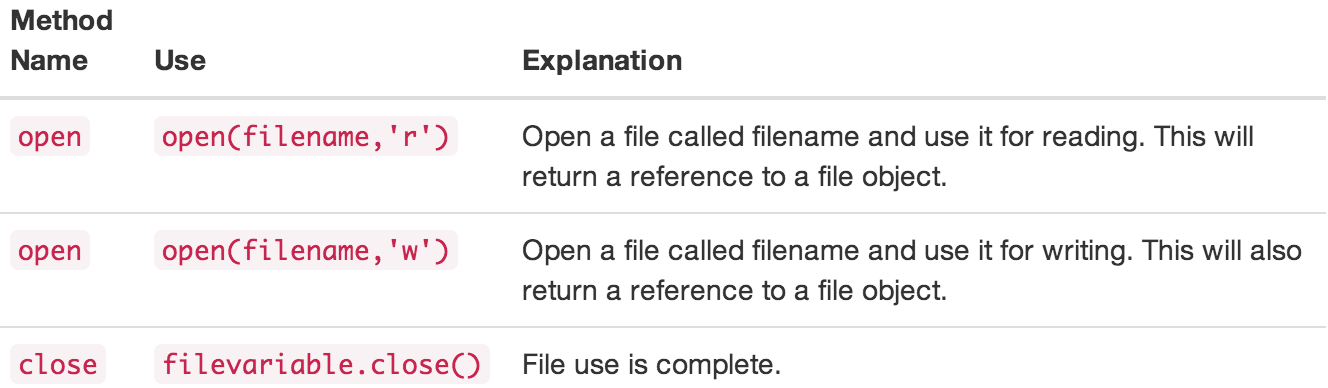
**flow of execution -** The order in which statements are executed during a program run.

**Function -** A named sequence of statements that performs some useful operation. Functions may or may not take parameters and may or may not produce a result.

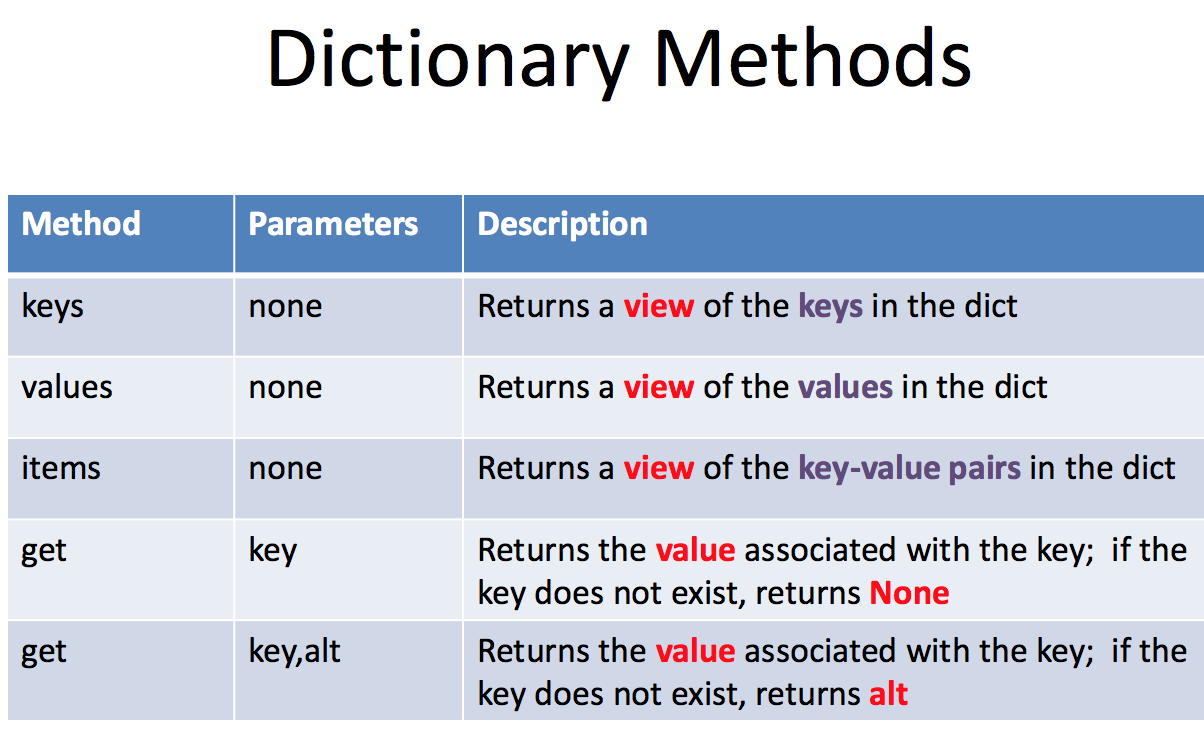
**function call -** A statement that executes a function. It consists of the name of the function followed by a list of arguments enclosed in parentheses.

**function composition -** Using the output from one function call as the input to another.

**function definition -** A statement that creates a new function, specifying its name, parameters, and the statements it executes.

**fruitful function -** A function that returns a value when it is called.

**header line -** The first part of a compound statement. A header line begins with a keyword and ends with a colon (:)

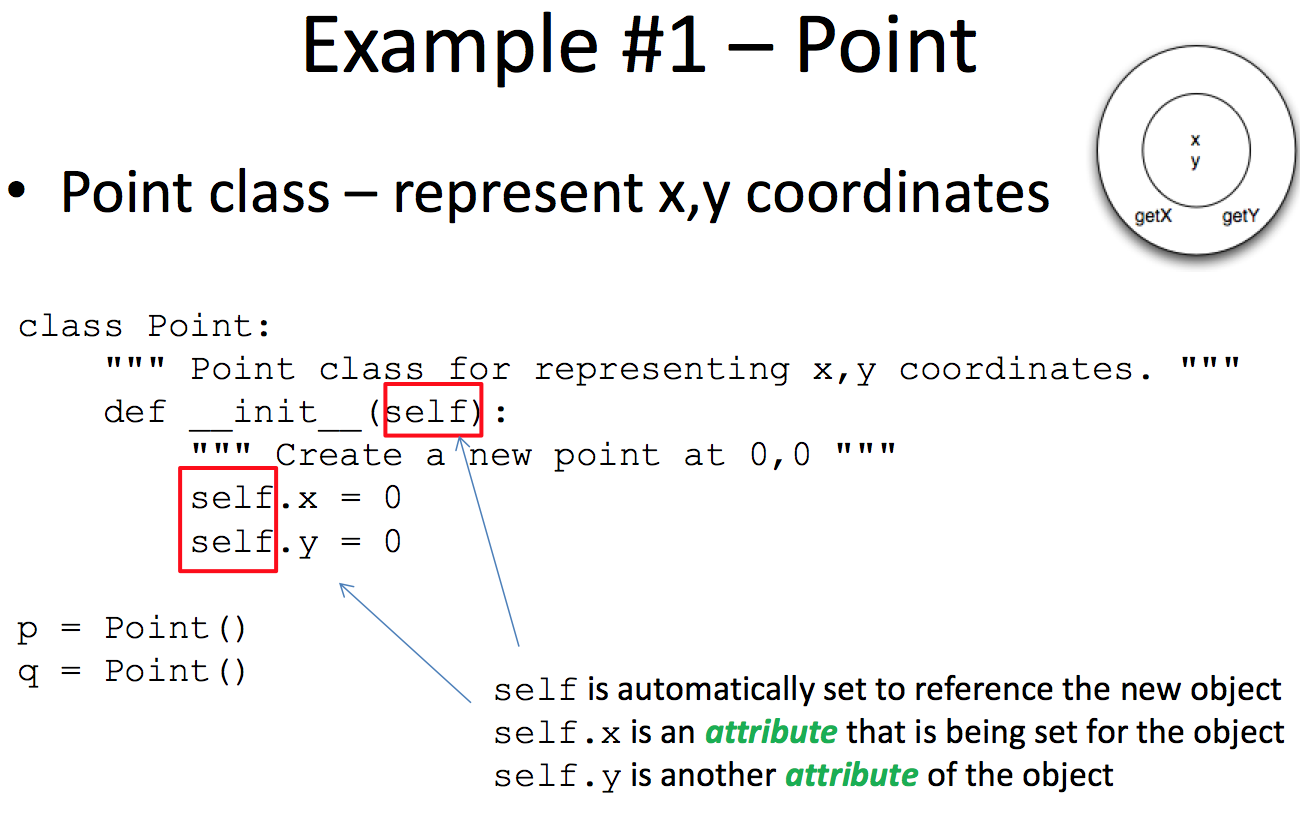
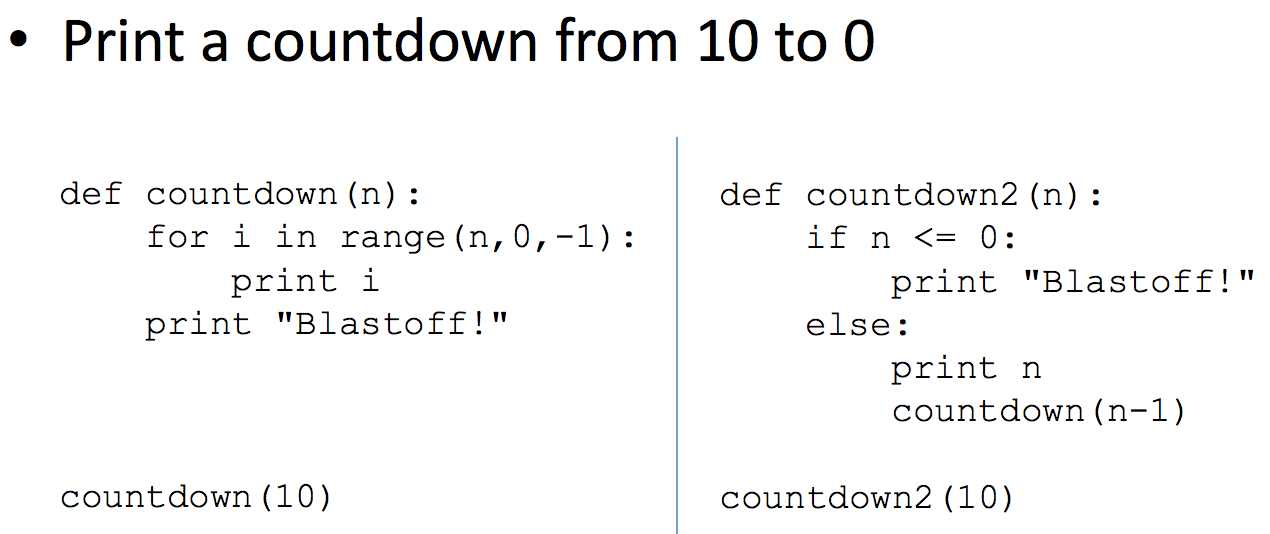
**import statement -** A statement which permits functions and variables defined in another Python module to be brought into the environment of another script. To use the features of the turtle, we need to first import the turtle module.

**Lifetime -** Variables and objects have lifetimes — they are created at some point during program execution, and will be destroyed at some time.

**local variable -** A variable defined inside a function. A local variable can only be used inside its function. Parameters of a function are also a special kind of local variable.

**Parameter -** A name used inside a function to refer to the value which was passed to it as an argument.

**Refactor -** A fancy word to describe reorganizing your program code, usually to make it more understandable. Typically, we have a program that is already working, then we go back to “tidy it up”. It often involves choosing better variable names, or spotting repeated patterns and moving that code into a function.

**Generalize -** To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter). Generalization makes code more versatile, more likely to be reused, and sometimes even easier to write.

Algorithm – step by step process

**indefinite iteration -** A loop where we just need to keep going until some condition is met. A while statement is used for this case.

def is\_prime(n):

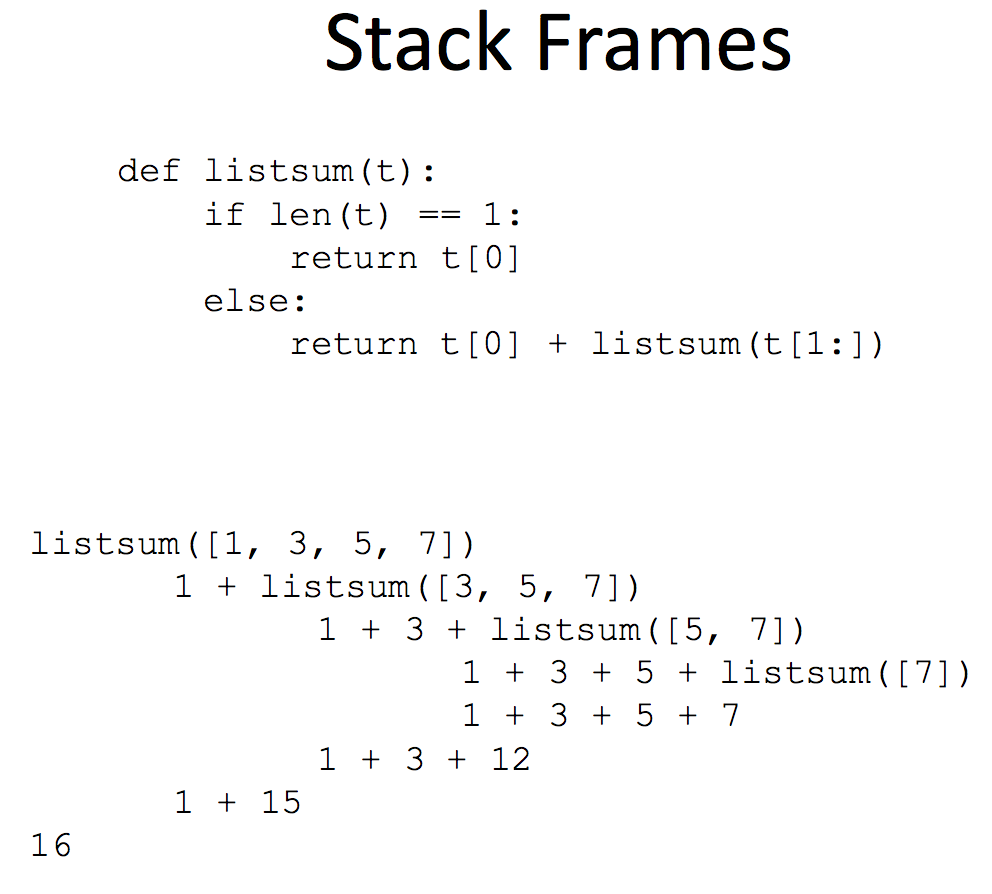
for i in range(2, n):

if n % i == 0:

return False

return True

Immutable *means that you can’t alter the string – can still index things from it*

**Constructor -** Every class has a “factory”, called by the same name as the class, for making new instances. If the class has an *initializer method*, this method is used to get the attributes (i.e. the state) of the new object properly set up.

def sameFraction(f1,f2):

return (f1.getNum() == f2.getNum()) and (f1.getDen() == f2.getDen())

class Fraction:

def \_\_init\_\_(self,top,bottom):

self.num = top #the numerator is on top

self.den = bottom #the denominator is on the bottom

def \_\_str\_\_(self):

return str(self.num) + "/" + str(self.den)

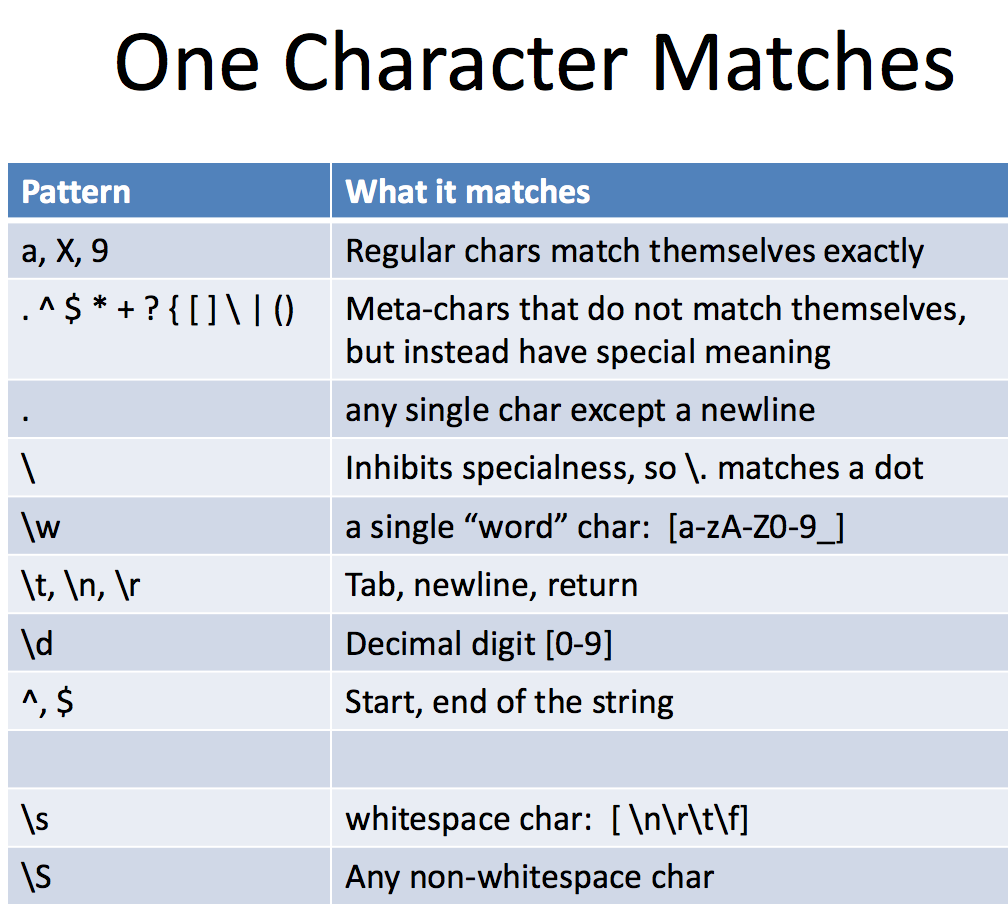
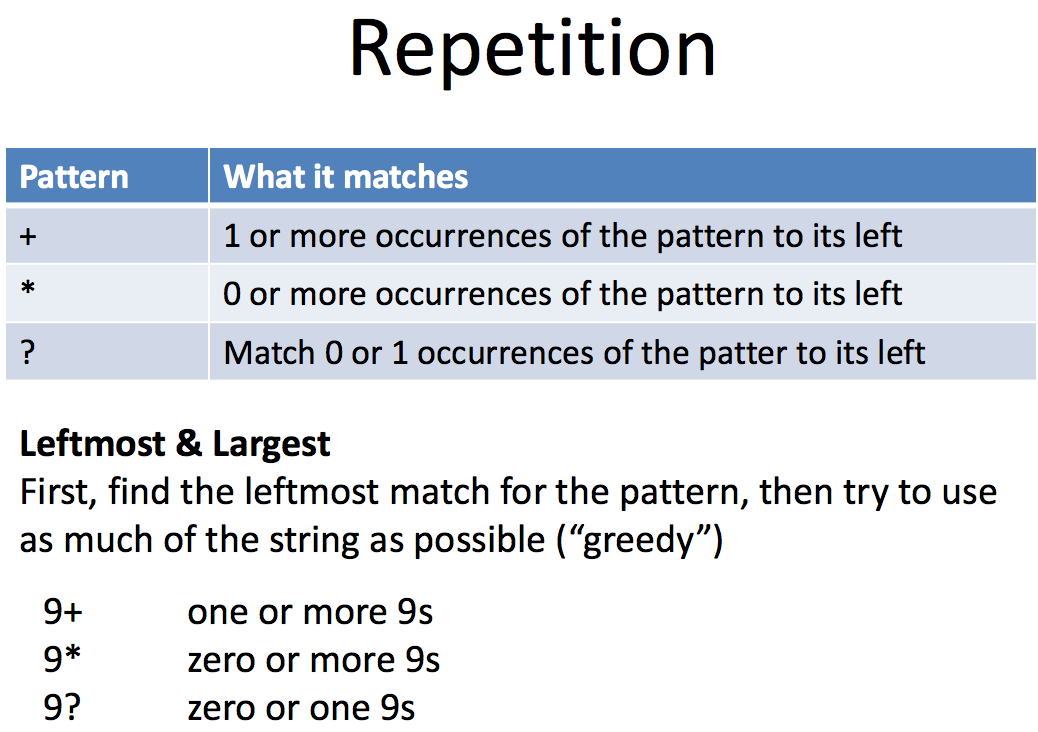
def getNum(self):

return self.num

def getDen(self):

return self.den

myfraction = Fraction(3,4)

yourfraction = Fraction(3,4)

print(myfraction is yourfraction) #false

print(sameFraction(myfraction,yourfraction)) #true

**deep copy -** To copy the contents of an object as well as any embedded objects, and any objects embedded in them, and so on; implemented by the deepcopy function in the copy module.

**deep equality -** Equality of values, or two references that point to objects that have the same value.

**shallow copy -** To copy the contents of an object, including any references to embedded objects; implemented by thecopy function in the copy module.

**shallow equality -** Equality of references, or two references that point to the same object.

