Introduction to Computation and Programming Using Python

Introduction to Python

i+j is the sum of i and j. If i and j are both of type int, the result is an int. If either of them is a float, the result is a float.

i–j is i minus j. If i and j are both of type int, the result is an int. If either of them is a float, the result is a float.

i*j is the product of i and j. If i and j are both of type int, the result is an int. If either of them is a float, the result is a float.

i//j is integer division. For example, the value of 6//2 is the int 3 and the value of 6//4 is the int 1. The value is 1 because integer division returns the quotient and ignores the remainder. If j == 0, an error occurs.

i/j is i divided by j. In Python 3, the / operator, performs floating point division. For example, the value of 6/4 is 1.5. If j == 0, an error occurs. (In Python 2, when i and j are both of type int, the / operator behaves the same way as // and returns an int. If either i or j is a float, it behaves like the Python 3 / operator.)

i%j is the remainder when the int i is divided by the int j. It is typically pronounced "i mod j," which is short for "i modulo j."

i**j is i raised to the power j. If i and j are both of type int, the result is an int. If either of them is a float, the result is a float.

The comparison operators are == (equal), != (not equal), > (greater), >= (at least), <, (less) and <= (at most).

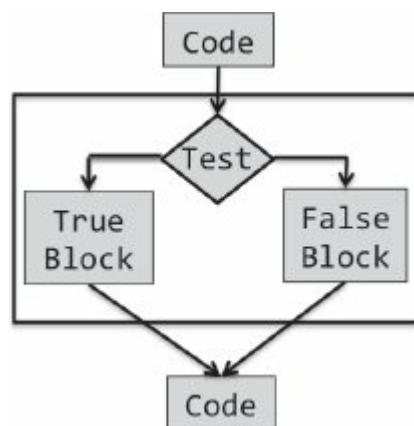*Figure - Operators on types int and float*

Figure - Flow chart for conditional statement

In Python, a conditional statement has the form
if *Boolean expression*:
    block of code
else:
    block of code
or
if *Boolean expression*:
    block of code

Some simple numerical programs

The code in Figure reimplements the exhaustive enumeration algorithm for finding cube roots. The break statement in the for loop causes the loop to terminate before it has been run on each element in the sequence over which it is iterating.

```python
#Find the cube root of a perfect cube
x = int(input('Enter an integer: '))
for ans in range(0, abs(x)+1):
    if ans**3 >= abs(x):
        break
if ans**3 != abs(x):
    print(x, 'is not a perfect cube')
else:
    if x < 0:
        ans = -ans
    print('Cube root of', x,'is', ans)
```

Figure - *Using for and break statements*

Figure contains code illustrating how to use this idea to quickly find an approximation to the square root.

```python
#Newton-Raphson for square root
#Find x such that x**2 - 24 is within epsilon of 0
epsilon = 0.01
k = 24.0
guess = k/2.0
while abs(guess*guess - k) >= epsilon:
    guess = guess - (((guess**2) - k)/(2*guess))
print('Square root of', k, 'is about', guess)
```

Figure *Implementation of Newton-Raphson method*

Functions, scoping, and abstraction

```python
def findRoot(x, power, epsilon):
    """Assumes x and epsilon int or float, power an int,
          epsilon > 0 & power >= 1
       Returns float y such that y**power is within epsilon of x.
          If such a float does not exist, it returns None"""
    if x < 0 and power%2 == 0: #Negative number has no even-powered
                                #roots
        return None
    low = min(-1.0, x)
    high = max(1.0, x)
    ans = (high + low)/2.0
    while abs(ans**power - x) >= epsilon:
        if ans**power < x:
            low = ans
        else:
            high = ans
        ans = (high + low)/2.0
    return ans

def testFindRoot():
    epsilon = 0.0001
    for x in [0.25, -0.25, 2, -2, 8, -8]:
        for power in range(1, 4):
            print('Testing x =', str(x), 'and power = ', power)
            result = findRoot(x, power, epsilon)
            if result == None:
                print('   No root')
            else:
                print('   ', result**power, '~=', x)
```

Figure - Finding an approximation to a root

open(fn, 'w') fn is a string representing a file name. Creates a file for writing and returns a file handle.

open(fn, 'r') fn is a string representing a file name. Opens an existing file for reading and returns a file handle.

open(fn, 'a') fn is a string representing a file name. Opens an existing file for appending and returns a file handle.

fh.read() returns a string containing the contents of the file associated with the file handle fh.

fh.readline() returns the next line in the file associated with the file handle fh.

fh.readlines() returns a list each element of which is one line of the file associated with the file handle fh.

fh.write(s) writes the string s to the end of the file associated with the file handle fh.

fh.writeLines(S) S is a sequence of strings. Writes each element of S as a separate line to the file associated with the file handle fh.

fh.close() closes the file associated with the file handle fh.

Figure - Common functions for accessing files

Structured types, mutability, and higherorder functions

seq[i] returns the ith element in the sequence.

len(seq) returns the length of the sequence.

seq1 + seq2 returns the concatenation of the two sequences (not available for ranges).

n*seq returns a sequence that repeats seq n times (not available for ranges).

seq[start:end] returns a slice of the sequence.

e in seq is True if e is contained in the sequence and False otherwise.

e not in seq is True if e is not in the sequence and False otherwise.

for e in seq iterates over the elements of the sequence.

Figure - Common operations on sequence types

Type	Type of elements	Examples of literals	Mutable
str	characters	'', 'a', 'abc'	No
tuple	any type	(), (3,), ('abc', 4)	No
range	integers	range(10), range(1, 10, 2)	No
list	any type	[], [3], ['abc', 4]	Yes

Figure - Comparison of sequence types

Testing and debugging

```python
def isPal(x):
    """Assumes x is a list
        Returns True if the list is a palindrome; False otherwise"""
    temp = x
    temp.reverse
    if temp == x:
        return True
    else:
        return False

def silly(n):
    """Assumes n is an int > 0
        Gets n inputs from user
        Prints 'Yes' if the sequence of inputs forms a palindrome;
            'No' otherwise"""
    for i in range(n):
        result = []
        elem = input('Enter element: ')
        result.append(elem)
    if isPal(result):
        print('Yes')
    else:
        print('No')
```

Figure Program with bugs

**MODERNISATION OF HIGHER EDUCATION IN CENTRAL
ASIA THROUGH NEW TECHNOLOGIES (HiEdTec)**

Co-funded by the
Erasmus+ Programme
of the European Union

This subsection contains a few pragmatic hints about what do when the debugging gets tough.

Look for the usual suspects. E.g., have you

- Passed arguments to a function in the wrong order,
- Misspelled a name, e.g., typed a lowercase letter when you should have typed an uppercase
- one,
- Failed to reinitialize a variable,
- Tested that two floating point values are equal (==) instead of nearly equal (remember that
- floating point arithmetic is not the same as the arithmetic you learned in school),
- Tested for value equality (e.g., compared two lists by writing the expression L1 == L2)
- when you meant object equality (e.g., id(L1) == id(L2)),
- Forgotten that some built-in function has a side effect,
- Forgotten the () that turns a reference to an object of type function into a function
- invocation,
- Created an unintentional alias, or
- Made any other mistake that is typical for you.

Stop asking yourself why the program isn't doing what you want it to. Instead, ask yourself why it is doing what it is. That should be an easier question to answer, and will probably be a good first step in figuring out how to fix the program.

Keep in mind that the bug is probably not where you think it is. If it were, you would probably have found it long ago. One practical way to go about deciding where to look is asking where the bug cannot be. As Sherlock Holmes said, "Eliminate all other factors, and the one which remains must be the truth."

Try to explain the problem to somebody else. We all develop blind spots. It is often the case that merely attempting to explain the problem to someone will lead you to see things you have missed. A good thing to try to explain is why the bug cannot be in certain places.

Don't believe everything you read. In particular, don't believe the documentation. The code may not be doing what the comments suggest.

Stop debugging and start writing documentation. This will help you approach the problem from a different perspective.

Walk away, and try again tomorrow. This may mean that bug is fixed later in time than if you had stuck with it, but you will probably spend a lot less of your time looking for it. That is, it is possible to trade latency for efficiency. (Students, this is an excellent reason to start work on programming problem sets earlier rather than later!)

Exceptions and assertions

```python
def getRatios(vect1, vect2):
    """Assumes: vect1 and vect2 are equal length lists of numbers
       Returns: a list containing the meaningful values of
            vect1[i]/vect2[i]"""
    ratios = []
    for index in range(len(vect1)):
        try:
            ratios.append(vect1[index]/vect2[index])
        except ZeroDivisionError:
            ratios.append(float('nan')) #nan = Not a Number
        except:
            raise ValueError('getRatios called with bad arguments')
    return ratios
```

Figure *Using exceptions for control flow*

In many programming languages, the standard approach to dealing with errors is to have functions return a value (often something analogous to Python's None) indicating that something has gone amiss.

Each function invocation has to check whether that value has been returned. In Python, it is more usual to have a function raise an exception when it cannot produce a result that is consistent with the function's specification.

The Python **raise** statement forces a specified exception to occur. The form of a raise statement is
raise exceptionName(*arguments*)

The *exceptionName* is usually one of the built-in exceptions, e.g., ValueError.

Classes and object-oriented programming

```python
class IntSet(object):
    """An intSet is a set of integers"""
    #Information about the implementation (not the abstraction)
    #Value of the set is represented by a list of ints, self.vals.
    #Each int in the set occurs in self.vals exactly once.

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if e not in self.vals:
            self.vals.append(e)

    def member(self, e):
        """Assumes e is an integer
           Returns True if e is in self, and False otherwise"""
        return e in self.vals

    def remove(self, e):
        """Assumes e is an integer and removes e from self
           Raises ValueError if e is not in self"""
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')

    def getMembers(self):
        """Returns a list containing the elements of self.
           Nothing can be assumed about the order of the elements"""
        return self.vals[:]

    def __str__(self):
        """Returns a string representation of self"""
        self.vals.sort()
        result = ''
        for e in self.vals:
            result = result + str(e) + ','
        return '{' + result[:-1] + '}' #-1 omits trailing comma
```

Figure *Class IntSet*

In Python, one implements data abstractions using **classes**. Figure contains a **class definition** that provides a straightforward implementation of a set-of-integers abstraction called IntSet.

A class definition creates an object of type type and associates with that class object a set of objects of type instancemethod. For example, the expression IntSet.insert refers to the method insert defined in the definition of the class IntSet. And the code print(type(IntSet), type(IntSet.insert))

A simplistic introduction to algorithmic complexity

Some of the most common instances of Big O are listed below. In each case, n is a measure of the size of the inputs to the function.

$O(1)$ denotes **constant** running time.

$O(\log n)$ denotes **logarithmic** running time.

$O(n)$ denotes **linear** running time.

$O(n \log n)$ denotes **log-linear** running time.

$O(n^k)$ denotes **polynomial** running time. Notice that k is a constant.

$O(c^n)$ denotes **exponential** running time. Here a constant is being raised to a power based on the size of the input.

Some simple algorithms and data structures

Figure defines two ordering functions, and then uses these to sort a list in two different ways. Each function uses the split method of type str.

When the code in Figure 10.6 is run, it prints

Sorted by last name = ['Tom Brady', 'Gisele Bundchen', 'Eric Grimson']

Sorted by first name = ['Eric Grimson', 'Gisele Bundchen', 'Tom Brady']

```python
def lastNameFirstName(name1, name2):
    arg1 = name1.split(' ')
    arg2 = name2.split(' ')
    if arg1[1] != arg2[1]:
        return arg1[1] < arg2[1]
    else: #last names the same, sort by first name
        return arg1[0] < arg2[0]

def firstNameLastName(name1, name2):
    arg1 = name1.split(' ')
    arg2 = name2.split(' ')
    if arg1[0] != arg2[0]:
        return arg1[0] < arg2[0]
    else: #first names the same, sort by last name
        return arg1[1] < arg2[1]

L = ['Tom Brady', 'Eric Grimson', 'Gisele Bundchen']
newL = mergeSort(L, lastNameFirstName)
print('Sorted by last name =', newL)
newL = mergeSort(L, firstNameLastName)
print('Sorted by first name =', newL)
```

Figure *Sorting a list of names*

The sorting algorithm used in most Python implementations is called **timsort**. The key idea is to take advantage of the fact that in a lot of data sets the data is already partially sorted. **Timsort's** worst-case performance is the same as merge sort's, but on average it performs considerably better.

As mentioned earlier, the Python method list.sort takes a list as its first argument and modifies that list. In contrast, the Python function sorted takes an iterable object (e.g., a list or a view) as its first argument and returns a new sorted list. For example, the code

```
L = [3,5,2]
D = {'a':12, 'c':5, 'b':'dog'}
print(sorted(L)) print(L) L.sort() print(L)
print(sorted(D)) D.sort()
will print
[2, 3, 5]
[3, 5, 2]
[2, 3, 5]
['a', 'b', 'c']
AttributeError: 'dict' object has no attribute
'sort'
```

Notice that when the sorted function is applied to a dictionary, it returns a sorted list of the keys of the dictionary. In contrast, when the sort method is applied to a dictionary, it causes an exception to be raised since there is no method dict.sort.

Plotting and more about classes

```python
class Mortgage(object):
    """Abstract class for building different kinds of mortgages"""
    def __init__(self, loan, annRate, months):
        self.loan = loan
        self.rate = annRate/12.0
        self.months = months
        self.paid = [0.0]
        self.outstanding = [loan]
        self.payment = findPayment(loan, self.rate, months)
        self.legend = None #description of mortgage

    def makePayment(self):
        self.paid.append(self.payment)
        reduction = self.payment - self.outstanding[-1]*self.rate
        self.outstanding.append(self.outstanding[-1] - reduction)

    def getTotalPaid(self):
        return sum(self.paid)
    def __str__(self):
        return self.legend

    def plotPayments(self, style):
        pylab.plot(self.paid[1:], style, label = self.legend)

    def plotBalance(self, style):
        pylab.plot(self.outstanding, style, label = self.legend)

    def plotTotPd(self, style):
        totPd = [self.paid[0]]
        for i in range(1, len(self.paid)):
            totPd.append(totPd[-1] + self.paid[i])
        pylab.plot(totPd, style, label = self.legend)

    def plotNet(self, style):
        totPd = [self.paid[0]]
        for i in range(1, len(self.paid)):
            totPd.append(totPd[-1] + self.paid[i])
        equityAcquired = pylab.array([self.loan] * \
                                     len(self.outstanding))
        equityAcquired = equityAcquired - \
                         pylab.array(self.outstanding)
        net = pylab.array(totPd) - equityAcquired
        pylab.plot(net, style, label = self.legend)
```

Figure Class Mortgage with plotting methods

Figure enhances class Mortgage by adding methods that make it convenient to produce such plots.

The nontrivial methods in class Mortgage are plotTotPd and plotNet. The method plotTotPd simply plots the cumulative total of the payments made. The method plotNet plots an approximation to the total cost of the mortgage over time by plotting the cash expended minus the equity acquired by paying off part of the loan.

The expression pylab.array(self.outstanding) in the function plotNet performs a type conversion. Thus far, we have been calling the plotting functions of PyLab with arguments of type list. Under the covers, PyLab has been converting these lists to a different type, array, which PyLab inherits from numpy.

Knapsack and graph optimization problems

```
def chooseBest(pset, maxWeight, getVal, getWeight):
    bestVal = 0.0
    bestSet = None
    for items in pset:
        itemsVal = 0.0
        itemsWeight = 0.0
        for item in items:
            itemsVal += getVal(item)
            itemsWeight += getWeight(item)
        if itemsWeight <= maxWeight and itemsVal > bestVal:
            bestVal = itemsVal
            bestSet = items
    return (bestSet, bestVal)

def testBest(maxWeight = 20):
    items = buildItems()
    pset = genPowerset(items)
    taken, val = chooseBest(pset, maxWeight, Item.getValue,
                            Item.getWeight)
    print('Total value of items taken is', val)
    for item in taken:
        print(item)
```

Figure *Brute-force optimal solution to the 0/1 knapsack problem*

Figure contains a straightforward implementation of this bruteforce approach to solving the 0/1 knapsack problem.

Dynamic programming

The code in Figure exploits the optimal substructure and overlapping subproblems to provide a dynamic programming solution to the 0/1 knapsack problem. An extra parameter, memo, has been added to keep track of solutions to subproblems that have already been solved. It is implemented using a dictionary with a key constructed from the length of toConsider and the available weight. The expression len(toConsider) is a compact way of representing the items still to be considered. This works because items are always removed from the same end (the front) of the list toConsider.

```python
def fastMaxVal(toConsider, avail, memo = {}):
    """Assumes toConsider a list of items, avail a weight
        memo supplied by recursive calls
      Returns a tuple of the total value of a solution to the
        0/1 knapsack problem and the items of that solution"""
    if (len(toConsider), avail) in memo:
        result = memo[(len(toConsider), avail)]
    elif toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0].getWeight() > avail:
        #Explore right branch only
        result = fastMaxVal(toConsider[1:], avail, memo)
    else:
        nextItem = toConsider[0]
        #Explore left branch
        withVal, withToTake =\
                fastMaxVal(toConsider[1:],
                          avail - nextItem.getWeight(), memo)
        withVal += nextItem.getValue()
        #Explore right branch
        withoutVal, withoutToTake = fastMaxVal(toConsider[1:],
                                              avail, memo)

        #Choose better branch
        if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    memo[(len(toConsider), avail)] = result
    return result
```

Figure *Dynamic programming solution to knapsack problem*

Random walks and more about data visualization

```python
def traceWalk(drunkKinds, numSteps):
    styleChoice = styleIterator(('k+', 'r^', 'mo'))
    f = Field()
    for dClass in drunkKinds:
        d = dClass()
        f.addDrunk(d, Location(0, 0))
        locs = []
        for s in range(numSteps):
            f.moveDrunk(d)
            locs.append(f.getLoc(d))
        xVals, yVals = [], []
        for loc in locs:
            xVals.append(loc.getX())
            yVals.append(loc.getY())
        curStyle = styleChoice.nextStyle()
        pylab.plot(xVals, yVals, curStyle,
                   label = dClass.__name__)
    pylab.title('Spots Visited on Walk ('
                + str(numSteps) + ' steps)')
    pylab.xlabel('Steps East/West of Origin')
    pylab.ylabel('Steps North/South of Origin')
    pylab.legend(loc = 'best')

traceWalk((UsualDrunk, ColdDrunk, EWDrunk), 200)
```
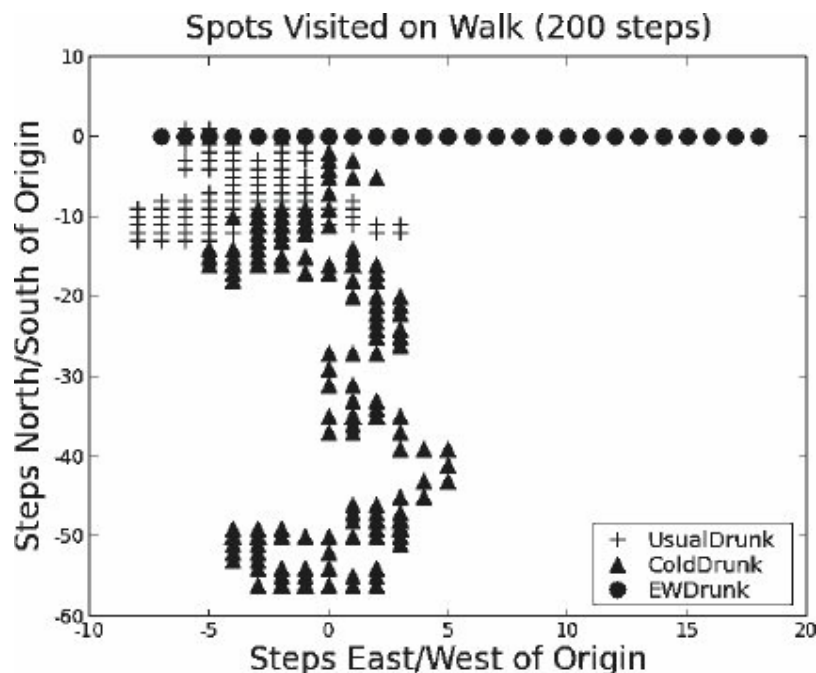
Figure *Tracing walks*

Figure *Trajectory of walks*

Stochastic programs, probability, and distributions

Figure 1 contains a function, flipPlot, that produces two plots, Figure 2, intended to show the law of large numbers at work. The first plot shows how the absolute value of the difference between the number of heads and number of tails changes with the number of flips. The second plot compares the ratio of heads to tails versus the number of flips. The line random.seed(0) near the bottom ensures that the pseudorandom number generator used by random.random will generate the same sequence of pseudorandom numbers each time this code is executed. This is convenient for debugging. The function random.seed can be called with any number. If it is called with no argument, the seed is chosen at random.

```python
def flipPlot(minExp, maxExp):
    """Assumes minExp and maxExp positive integers; minExp < maxExp
       Plots results of 2**minExp to 2**maxExp coin flips"""
    ratios, diffs, xAxis = [], [], []
    for exp in range(minExp, maxExp + 1):
        xAxis.append(2**exp)
    for numFlips in xAxis:
        numHeads = 0
        for n in range(numFlips):
            if random.choice(('H', 'T')) == 'H':
                numHeads += 1
        numTails = numFlips - numHeads
        try:
            ratios.append(numHeads/numTails)
            diffs.append(abs(numHeads - numTails))
        except ZeroDivisionError:
            continue
    pylab.title('Difference Between Heads and Tails')
    pylab.xlabel('Number of Flips')
    pylab.ylabel('Abs(#Heads - #Tails)')
    pylab.plot(xAxis, diffs, 'k')
    pylab.figure()
    pylab.title('Heads/Tails Ratios')
    pylab.xlabel('Number of Flips')
    pylab.ylabel('#Heads/#Tails')
    pylab.plot(xAxis, ratios, 'k')

random.seed(0)
flipPlot(4, 20)
```
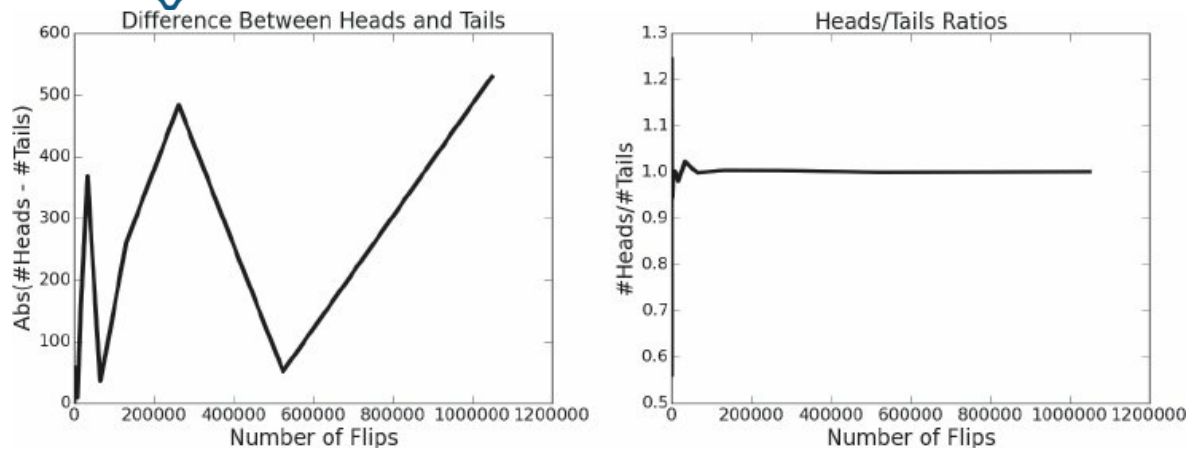
Figure 1 Plotting the results of coin flips

Figure 2 *The law of large numbers at work*

Understanding experimental data

```python
def getTrajectoryData(fileName):
    dataFile = open(fileName, 'r')
    distances = []
    heights1, heights2, heights3, heights4 = [],[],[],[]
    dataFile.readline()
    for line in dataFile:
        d, h1, h2, h3, h4 = line.split()
        distances.append(float(d))
        heights1.append(float(h1))
        heights2.append(float(h2))
        heights3.append(float(h3))
        heights4.append(float(h4))
    dataFile.close()
    return (distances, [heights1, heights2, heights3, heights4])

def processTrajectories(fileName):
    distances, heights = getTrajectoryData(fileName)
    numTrials = len(heights)
    distances = pylab.array(distances)
    #Get array containing mean height at each distance
    totHeights = pylab.array([0]*len(distances))
    for h in heights:
        totHeights = totHeights + pylab.array(h)
    meanHeights = totHeights/len(heights)
    pylab.title('Trajectory of Projectile (Mean of '\
                + str(numTrials) + ' Trials)')
    pylab.xlabel('Inches from Launch Point')
    pylab.ylabel('Inches Above Launch Point')
    pylab.plot(distances, meanHeights, 'ko')
    fit = pylab.polyfit(distances, meanHeights, 1)
    altitudes = pylab.polyval(fit, distances)
    pylab.plot(distances, altitudes, 'b', label = 'Linear Fit')
    fit = pylab.polyfit(distances, meanHeights, 2)
    altitudes = pylab.polyval(fit, distances)
    pylab.plot(distances, altitudes, 'k:', label = 'Quadratic Fit')
    pylab.legend()

processTrajectories('launcherData.txt')
```

Figure *Plotting the trajectory of a projectile*

MODERNISATION OF HIGHER EDUCATION IN CENTRAL ASIA THROUGH NEW TECHNOLOGIES (HiEdTec)

Co-funded by the
Erasmus+ Programme
of the European Union

The code in Figure was used to plot the mean altitude of the projectile in the four trials against the distance from the point of launch.

It also plots the best linear and quadratic fits to those points. (In case you have forgotten the meaning of multiplying a list by an integer, the expression [0]*len(distances) produces a list of len(distances) 0's.)

Clustering

K-means clustering is probably the most widely used clustering method. Its goal is to partition a set of examples into k clusters such that
- Each example is in the cluster whose centroid is the closest centroid to that example, and
- The dissimilarity of the set of clusters is minimized.

Figure contains a function, trykmeans, that calls kmeans multiple times and selects the result with the lowest dissimilarity. If a trial fails because kmeans generated an empty cluster and therefore raised an exception, trykmeans merely tries again-assuming that eventually kmeans will choose an initial set of centroids that successfully converges.

```python
def dissimilarity(clusters):
    totDist = 0.0
    for c in clusters:
        totDist += c.variability()
    return totDist

def trykmeans(examples, numClusters, numTrials, verbose = False):
    """Calls kmeans numTrials times and returns the result with the
        lowest dissimilarity"""
    best = kmeans(examples, numClusters, verbose)
    minDissimilarity = dissimilarity(best)
    trial = 1
    while trial < numTrials:
        try:
            clusters = kmeans(examples, numClusters, verbose)
        except ValueError:
            continue #If failed, try again
        currDissimilarity = dissimilarity(clusters)
        if currDissimilarity < minDissimilarity:
            best = clusters
            minDissimilarity = currDissimilarity
        trial += 1
    return best
```

***Figure** Finding the best k-means clustering*