The following program calculates all Pythagorean numbers less than a maximal number.   
Remark: We have to import the math module to be able to calculate the square root of a number. 

#!/usr/bin/env python

from math import sqrt

n = raw\_input("Maximal Number? ")

n = int(n)+1

for a in range(1,n):

for b in range(a,n):

c\_square = a\*\*2 + b\*\*2

c = int(sqrt(c\_square))

if ((c\_square - c\*\*2) == 0):

print a, b, c

===========================================================================

>>> dishes = ["pizza", "sauerkraut", "paella", "Hamburger"]

>>> countries = ["Italy", "Germany", "Spain", "USA"]

>>> country\_specialities = zip(countries, dishes)

>>> print country\_specialities

[('Italy', 'pizza'), ('Germany', 'sauerkraut'), ('Spain', 'paella'), ('USA', 'Hamburger')]

>>>

>>> country\_specialities\_dict = dict(country\_specialities)

>>> print country\_specialities\_dict

{'Germany': 'sauerkraut', 'Spain': 'paella', 'Italy': 'pizza', 'USA': 'Hamburger'}

>>>

There is still one question concerning the function zip(). What happens, if one of the two argument lists contains more elements than the other one? It's easy: The superfluous elements will not be used:

>>> countries = ["Italy", "Germany", "Spain", "USA", "Switzerland"]

>>> dishes = ["pizza", "sauerkraut", "paella", "Hamburger"]

>>> country\_specialities = zip(countries,dishes)

>>> print country\_specialities

[('Italy', 'pizza'), ('Germany', 'sauerkraut'), ('Spain', 'paella'), ('USA', 'Hamburger')]

If we want to create a set, we can call the built-in set function with a sequence or another iterable object.   
  
In the following example, a string is singularized into its characters to build the resulting set x:

>>> x = set("A Python Tutorial")

>>> x

set(['A', ' ', 'i', 'h', 'l', 'o', 'n', 'P', 'r', 'u', 't', 'a', 'y', 'T'])

>>> type(x)

<type 'set'>

>>>

>>> cities = set(("Paris", "Lyon", "London","Berlin","Paris","Birmingham"))

>>> cities

set(['Paris', 'Birmingham', 'Lyon', 'London', 'Berlin'])

>>>

>>> cities = set((("Python","Perl"), ("Paris", "Berlin", "London")))

>>> cities = set((["Python","Perl"], ["Paris", "Berlin", "London"]))

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unhashable type: 'list'

>>>

Though sets can't contain mutable objects, sets are mutable: 

>>> cities = set(["Frankfurt", "Basel","Freiburg"])

>>> cities.add("Strasbourg")

>>> cities

set(['Freiburg', 'Basel', 'Frankfurt', 'Strasbourg'])

>>>

Frozensets are like sets except that they cannot be changed, i.e. they are immutable:

>>> cities = frozenset(["Frankfurt", "Basel","Freiburg"])

>>> cities.add("Strasbourg")

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'frozenset' object has no attribute 'add'

>>>

>>> adjectives = {"cheap","expensive","inexpensive","economical"}

>>> adjectives

set(['inexpensive', 'cheap', 'expensive', 'economical'])

>>>

>>> colours = {"red","green"}

>>> colours.add("yellow")

>>> colours

set(['green', 'yellow', 'red'])

>>> colours.add(["black","white"])

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unhashable type: 'list'

>>>

This method returns the difference of two or more sets as a new set.

>>> x = {"a","b","c","d","e"}

>>> y = {"b","c"}

>>> z = {"c","d"}

>>> x.difference(y)

set(['a', 'e', 'd'])

>>> x.difference(y).difference(z)

set(['a', 'e'])

>>>

Instead of using the method difference, we can use the operator "-":

>>> x - y

set(['a', 'e', 'd'])

>>> x - y - z

set(['a', 'e'])

>>>

Returns the intersection of the instance set and the set s as a new set. In other words: A set with all the elements which are contained in both sets is returned.

>>> x = {"a","b","c","d","e"}

>>> y = {"c","d","e","f","g"}

>>> x.intersection(y)

set(['c', 'e', 'd'])

>>>

This can be abbreviated with the ampersand operator "&":

>>> x = {"a","b","c","d","e"}

>>> y = {"c","d","e","f","g"}

>>> x.intersection(y)

set(['c', 'e', 'd'])

>>>

>>> x = {"a","b","c","d","e"}

>>> y = {"c","d","e","f","g"}

>>> x & y

set(['c', 'e', 'd'])

>>>

It's possible to completely copy shallow list structures with the slice operator without having any of the side effects, which we have described above:

>>> list1 = ['a','b','c','d']

>>> list2 = list1[:]

>>> list2[1] = 'x'

>>> print list2

['a', 'x', 'c', 'd']

>>> print list1

['a', 'b', 'c', 'd']

>>>

**Arbitrary Number of Parameters**

There are many situations in programming, in which the exact number of necessary parameters cannot be determined a-priori. An arbitrary parameter number can be accomplished in Python with so-called tuple references. An asterisk "\*" is used in front of the last parameter name to denote it as a tuple reference. This asterisk shouldn't be mistaken with the C syntax, where this notation is connected with pointers.   
Example:

def arbitrary(x, y, \*more):

print "x=", x, ", y=", y

print "arbitrary: ", more

x and y are regular positional parameters in the previous function. \*more is a tuple reference.   
Example:

>>> execfile("funktion1.py")

>>> arbitrary(3,4)

x= 3 , x= 4

arbitrary: ()

>>> arbitrary(3,4, "Hello World", 3 ,4)

x= 3 , x= 4

arbitrary: ('Hello World', 3, 4)

def factorial(n):

if n == 1:

return 1

else:

return n \* factorial(n-1)

We can track how the function works by adding two print() functions to the previous function definition:

def factorial(n):

print("factorial has been called with n = " + str(n))

if n == 1:

return 1

else:

res = n \* factorial(n-1)

print("intermediate result for ", n, " \* factorial(" ,n-1, "): ",res)

return res

print(factorial(5))

This Python script outputs the following results:

factorial has been called with n = 5

factorial has been called with n = 4

factorial has been called with n = 3

factorial has been called with n = 2

factorial has been called with n = 1

intermediate result for 2 \* factorial( 1 ): 2

intermediate result for 3 \* factorial( 2 ): 6

intermediate result for 4 \* factorial( 3 ): 24

intermediate result for 5 \* factorial( 4 ): 120

120

Let's have a look at an iterative version of the factorial function.

def iterative\_factorial(n):

result = 1

for i in range(2,n+1):

result \*= i

return result

def fib(n):

if n == 0:

return 0

elif n == 1:

return 1

else:

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

An iterative solution for the problem is also easy to write, though the recursive solution looks more like the mathematical definition:

def fibi(n):

a, b = 0, 1

for i in range(n):

a, b = b, a + b

return a

## Logarithmic complexity

One famous problem in computer science is that of searching for a value within an array. We solved this problem earlier for the general case. This problem becomes interesting if we have an array which is sorted and we want to find a given value within it. One method to do that is called binary search. We look at the middle element of our array: If we find it there, we're done. Otherwise, if the value we find there is bigger than the value we're looking for, we know that our element will be on the left part of the array. Otherwise, we know it'll be on the right part of the array. We can keep cutting these smaller arrays in halves until we have a single element to look at. Here's the method using pseudocode:

def binarySearch( A, n, value ):

if n = 1:

if A[ 0 ] = value:

return true

else:

return false

if value < A[ n / 2 ]:

return binarySearch( A[ 0...( n / 2 - 1 ) ], n / 2 - 1, value )

else if value > A[ n / 2 ]:

return binarySearch( A[ ( n / 2 + 1 )...n ], n / 2 - 1, value )

else:

return true

This pseudocode is a simplification of the actual implementation. In practice, this method is easier described than implemented, as the programmer needs to take care of some implementation issues. There are off-by-one errors and the division by 2 may not always produce an integer value and so it's necessary to floor() or ceil() the value. But we can assume for our purposes that it will always succeed, and we'll assume our actual implementation in fact takes care of the off-by-one errors, as we only want to analyze the complexity of this method. If you've never implemented binary search before, you may want to do this in your favourite programming language. It's a truly enlightening endeavor.

See **Figure 6** to help you understand the way binary search operates.

If you're unsure that this method actually works, take a moment now to run it by hand in a simple example and convince yourself that it actually works.

Let us now attempt to analyze this algorithm. Again, we have a recursive algorithm in this case. Let's assume, for simplicity, that the array is always cut in exactly a half, ignoring just now the + 1 and - 1 part in the recursive call. By now you should be convinced that a little change such as ignoring + 1 and - 1 won't affect our complexity results. This is a fact that we would normally have to prove if we wanted to be prudent from a mathematical point of view, but practically it is intuitively obvious. Let's assume that our array has a size that is an exact power of 2, for simplicity. Again this assumption doesn't change the final results of our complexity that we will arrive at. The worst-case scenario for this problem would happen when the value we're looking for does not occur in our array at all. In that case, we'd start with an array of size n in the first call of the recursion, then get an array of size n / 2 in the next call. Then we'll get an array of size n / 4 in the next recursive call, followed by an array of size n / 8 and so forth. In general, our array is split in half in every call, until we reach 1. So, let's write the number of elements in our array for every call:

1. 0th iteration: n
2. 1st iteration: n / 2
3. 2nd iteration: n / 4
4. 3rd iteration: n / 8
5. ...
6. ith iteration: n / 2i
7. ...
8. last iteration: 1

Notice that in the i-th iteration, our array has n / 2i elements. This is because in every iteration we're cutting our array into half, meaning we're dividing its number of elements by two. This translates to multiplying the denominator with a 2. If we do that i times, we get n / 2i. Now, this procedure continues and with every larger i we get a smaller number of elements until we reach the last iteration in which we have only 1 element left. If we wish to find i to see in what iteration this will take place, we have to solve the following equation:

1 = n / 2i

This will only be true when we have reached the final call to the binarySearch() function, not in the general case. So solving for i here will help us find in which iteration the recursion will finish. Multiplying both sides by 2i we get:

2i = n

Now, this equation should look familiar if you read the logarithms section above. Solving for i we have:

i = log( n )

This tells us that the number of iterations required to perform a binary search is log( n ) where n is the number of elements in the original array.

If you think about it, this makes some sense. For example, take n = 32, an array of 32 elements. How many times do we have to cut this in half to get only 1 element? We get: 32 → 16 → 8 → 4 → 2 → 1. We did this 5 times, which is the logarithm of 32. Therefore, the complexity of binary search is Θ( log( n ) ).

This last result allows us to compare binary search with linear search, our previous method. Clearly, as log( n ) is much smaller than n, it is reasonable to conclude that binary search is a much faster method to search within an array then linear search, so it may be advisable to keep our arrays sorted if we want to do many searches within them.

To perform a mergesort, we will first need to build a helper function that we will then use to do the actual sorting. We will m ake a merge function which takes two arrays that are both already sorted and merges them together into a big sorted array. This is easily done:

def merge( A, B ):

if empty( A ):

return B

if empty( B ):

return A

if A[ 0 ] < B[ 0 ]:

return concat( A[ 0 ], merge( A[ 1...A\_n ], B ) )

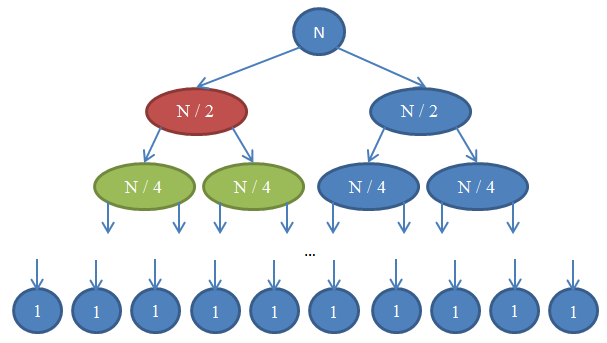
else:

return concat( B[ 0 ], merge( A, B[ 1...B\_n ] ) )

merge sort, binary search, linear, heapsort

Take a look at **Figure 7** to understand this recursion.

word memorization, though in a way it has something in common. Memoisation is a technique used in computing to speed up programs. This is accomplished by memorizing the calculation results of processed input such as the results of function calls. If the same input or a function call with the same parameters is used, the previously stored results can be used again and unnecessary calculation are avoided

**Figure 7**: The recursion tree of merge sort.

Let's see what's going on here. Each circle represents a call to the mergeSort function. The number written in the circle indicates the size of the array that is being sorted. The top blue circle is the original call to mergeSort, where we get to sort an array of size n. The arrows indicate recursive calls made between functions. The original call to mergeSort makes two calls to mergeSort on two arrays, each of size n / 2. This is indicated by the two arrows at the top. In turn, each of these calls makes two calls of its own to mergeSort two arrays of size n / 4 each, and so forth until we arrive at arrays of size 1. This diagram is called a recursion tree, because it illustrates how the recursion behaves and looks like a tree (the root is at the top and the leaves are at the bottom, so in reality it looks like an inversed tree).

Notice that at each row in the above diagram, the total number of elements is n. To see this, take a look at each row individually. The first row contains only one call to mergeSort with an array of size n, so the total number of elements is n. The second row has two calls to mergeSort each of size n / 2. But n / 2 + n / 2 = n and so again in this row the total number of elements is n. In the third row, we have 4 calls each of which is applied on an n / 4-sized array, yielding a total number of elements equal to n / 4 + n / 4 + n / 4 + n / 4 = 4n / 4 = n. So again we get n elements. Now notice that at each row in this diagram the caller will have to perform a merge operation on the elements returned by the callees. For example, the circle indicated with red color has to sort n / 2 elements. To do this, it splits the n / 2-sized array into two n / 4-sized arrays, calls mergeSortrecursively to sort those (these calls are the circles indicated with green color), then merges them together. This merge operation requires to merge n / 2 elements. At each row in our tree, the total number of elements merged is n. In the row that we just explored, our function merges n / 2 elements and the function on its right (which is in blue color) also has to merge n / 2 elements of its own. That yields n elements in total that need to be merged for the row we're looking at.

By this argument, the complexity for each row is Θ( n ). We know that the number of rows in this diagram, also called the depth of the recursion tree, will be log( n ). The reasoning for this is exactly the same as the one we used when analyzing the complexity of binary search. We have log( n ) rows and each of them is Θ( n ), therefore the complexity of mergeSortis Θ( n \* log( n ) ). This is much better than Θ( n2 ) which is what selection sort gave us (remember that log( n ) is much smaller than n, and so n \* log( n ) is much smaller than n \* n = n2). If this sounds complicated to you, don't worry: It's not easy the first time you see it. Revisit this section and reread about the arguments here after you implement mergesort in your favourite programming language and validate that it works.

As you saw in this last example, complexity analysis allows us to compare algorithms to see which one is better. Under these circumstances, we can now be pretty certain that merge sort will outperform selection sort for large arrays. This conclusion would be hard to draw if we didn't have the theoretical background of algorithm analysis that we developed. In practice, indeed sorting algorithms of running time Θ( n \* log( n ) ) are used. For example, [the Linux kernel uses a sorting algorithm called heapsort](http://lxr.free-electrons.com/source/lib/sort.c), which has the same running time as mergesort which we explored here, namely Θ( n log( n ) ) and so is optimal. Notice that we have not proven that these sorting algorithms are optimal. Doing this requires a slightly more involved mathematical argument, but rest assured that they can't get any better from a complexity point of view.

#### Functions as Parameters

If you solely look at the previous examples, this doesn't seem to be very usefull. It gets useful in combination with two further powerful possibilities of Python functions. Due to the fact that every parameter of a function is a reference to an object and functions are objects as well, we can pass functions - or better "references to functions" - as parameters to a function. We will demonstrate this in the next simple example:

def g():

print("Hi, it's me 'g'")

print("Thanks for calling me")

def f(func):

print("Hi, it's me 'f'")

print("I will call 'func' now")

func()

f(g)

The output looks like this:

Hi, it's me 'f'

I will call 'func' now

Hi, it's me 'g'

Thanks for calling me

formula for polynomials with degree 2   
  
The Python implementation as a polynomial factory function can be written like this:

def polynomial\_creator(a, b, c):

def polynomial(x):

return a \* x\*\*2 + b \* x + c

return polynomial

p1 = polynomial\_creator(2, 3, -1)

p2 = polynomial\_creator(-1, 2, 1)

for x in range(-2, 2, 1):

print(x, p1(x), p2(x))

We will rewrite now our initial example. Instead of writing the statement

foo = our\_decorator(foo)

we can write

@our\_decorator

But this line has to be directly positioned in front of the decorated function. The complete example looks like this now:

def our\_decorator(func):

def function\_wrapper(x):

print("Before calling " + func.\_\_name\_\_)

func(x)

print("After calling " + func.\_\_name\_\_)

return function\_wrapper

@our\_decorator

def foo(x):

print("Hi, foo has been called with " + str(x))

foo("Hi")

**Summarizing we can say that a decorator in Python is a callable Python object that is used to modify a function, method or class definition. The original object, the one which is going to be modified, is passed to a decorator as an argument. The decorator returns a modified object, e.g. a modified function, which is bound to the name used in the definition.**

The previous function\_wrapper works only for functions with exactly one parameter. We provide a generalized version of the function\_wrapper, which accepts functions with arbitrary parameters in the following example: 

from random import random, randint, choice

def our\_decorator(func):

def function\_wrapper(\*args, \*\*kwargs):

print("Before calling " + func.\_\_name\_\_)

res = func(\*args, \*\*kwargs)

print(res)

print("After calling " + func.\_\_name\_\_)

return function\_wrapper

random = our\_decorator(random)

randint = our\_decorator(randint)

choice = our\_decorator(choice)

random()

randint(3, 8)

choice([4, 5, 6])

The result looks as expected:

Before calling random

0.16420183945821654

After calling random

Before calling randint

8

After calling randint

Before calling choice

5

After calling choice

### Use Cases for Decorators

#### Checking Arguments with a Decorator

In our chapter about recursive functions we introduced the factorial function. We wanted to keep the function as simple as possible and we didn't want to obscure the underlying idea, so we hadn't incorporated any argument checks. So, if somebody had called our function with a negative argument or with a float argument, our function would have got into an endless loop.   
  
The following program uses a decorator function to ensure that the argument passed to the function factorial is a positive integer:

def argument\_test\_natural\_number(f):

def helper(x):

if type(x) == int and x > 0:

return f(x)

else:

raise Exception("Argument is not an integer")

return helper

@argument\_test\_natural\_number

def factorial(n):

if n == 1:

return 1

else:

return n \* factorial(n-1)

for i in range(1,10):

print(i, factorial(i))

print(factorial(-1))

### Decorators with Parameters

We define two decorators in the following code:

def evening\_greeting(func):

def function\_wrapper(x):

print("Good evening, " + func.\_\_name\_\_ + " returns:")

func(x)

return function\_wrapper

def morning\_greeting(func):

def function\_wrapper(x):

print("Good morning, " + func.\_\_name\_\_ + " returns:")

func(x)

return function\_wrapper

@evening\_greeting

def foo(x):

print(42)

foo("Hi")

These two decorators are nearly the same, except for the greeting. We want to add a parameter to the decorator to be capable of customizing the greeting, when we do the decoration. We have to wrap another function around our previous decorator function to accomplish this. We can now easy say "Good Morning" in the Greek way:

def greeting(expr):

def greeting\_decorator(func):

def function\_wrapper(x):

print(expr + ", " + func.\_\_name\_\_ + " returns:")

func(x)

return function\_wrapper

return greeting\_decorator

@greeting("ÎºÎ±Î»Î·Î¼ÎµÏÎ±")

def foo(x):

print(42)

foo("Hi")

The output:

ÎºÎ±Î»Î·Î¼ÎµÏÎ±, foo returns:

42

If we don't want or cannot use the "at" decorator syntax, we can do it with function calls:

def greeting(expr):

def greeting\_decorator(func):

def function\_wrapper(x):

print(expr + ", " + func.\_\_name\_\_ + " returns:")

func(x)

return function\_wrapper

return greeting\_decorator

def foo(x):

print(42)

greeting2 = greeting("ÎºÎ±Î»Î·Î¼ÎµÏÎ±")

foo = greeting2(foo)

foo("Hi")

Of course, we don't need the additional definition of "greeting2". We can directly apply the result of the call "greeting("ÎºÎ±Î»Î·Î¼ÎµÏÎ±")" on "foo":

foo = greeting("ÎºÎ±Î»Î·Î¼ÎµÏÎ±")(foo)

**Using wraps from functools**

The way we have defined decorators so far hasn't taken into account that the attributes

* \_\_name\_\_ (name of the function),
* \_\_doc\_\_ (the docstring) and
* \_\_module\_\_ (The module in which the function is defined)

of the original functions will be lost after the decoration.   
  
The following decorator will be saved in a file greeting\_decorator.py:

def greeting(func):

def function\_wrapper(x):

""" function\_wrapper of greeting """

print("Hi, " + func.\_\_name\_\_ + " returns:")

return func(x)

return function\_wrapper

We call it in the following program:

from greeting\_decorator import greeting

@greeting

def f(x):

""" just some silly function """

return x + 4

f(10)

print("function name: " + f.\_\_name\_\_)

print("docstring: " + f.\_\_doc\_\_)

print("module name: " + f.\_\_module\_\_)

We get the following "unwanted" results:

Hi, f returns:

function name: function\_wrapper

docstring: function\_wrapper of greeting

module name: greeting\_decorator

We can save the original attributes of the function f, if we assign them inside of the decorator. We change our previous decorator accordingly and save it as greeting\_decorator\_manually.py:

def greeting(func):

def function\_wrapper(x):

""" function\_wrapper of greeting """

print("Hi, " + func.\_\_name\_\_ + " returns:")

return func(x)

function\_wrapper.\_\_name\_\_ = func.\_\_name\_\_

function\_wrapper.\_\_doc\_\_ = func.\_\_doc\_\_

function\_wrapper.\_\_module\_\_ = func.\_\_module\_\_

return function\_wrapper

In our main program, all we have to do is change the import statement to

from greeting\_decorator\_manually import greeting

Now we get the proper results:

Hi, f returns:

function name: f

docstring: just some silly function

module name: \_\_main\_\_

Fortunately, we don't have to add all this code to our decorators to have these results. We can import the decorator "wraps" from functools instead and decorate our function in the decorator with it:

from functools import wraps

def greeting(func):

@wraps(func)

def function\_wrapper(x):

""" function\_wrapper of greeting """

print("Hi, " + func.\_\_name\_\_ + " returns:")

return func(x)

return function\_wrapper

### Classes instead of Functions

#### The \_\_call\_\_ method

So far we used functions as decorators. Before we can define a decorator as a class, we have to introduce the \_\_call\_\_ method of classes. We mentioned alreaedy that a decorator is simply a callable object that takes a function as an input parameter. A function is a callable object, but what lots of Python programmers don't know. We can define classes as callable objects as well. The \_\_call\_\_ method is called, if the instance is called "like a function", i.e. using brackets.

class A:

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

print("An instance of A was initialized")

def \_\_call\_\_(self, \*args, \*\*kwargs):

print("Arguments are:", args, kwargs)

x = A()

print("now calling the instance:")

x(3, 4, x=11, y=10)

print("Let's call it again:")

x(3, 4, x=11, y=10)

We get the following output:

An instance of A was initialized

now calling the instance:

Arguments are: (3, 4) {'x': 11, 'y': 10}

Let's call it again:

Arguments are: (3, 4) {'x': 11, 'y': 10}

We can write a class for the fibonacci function by using \_\_call\_\_:

class Fibonacci:

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

self.cache = {}

def \_\_call\_\_(self, n):

if n not in self.cache:

if n == 0:

self.cache[0] = 0

elif n == 1:

self.cache[1] = 1

else:

self.cache[n] = self.\_\_call\_\_(n-1) + self.\_\_call\_\_(n-2)

return self.cache[n]

fib = Fibonacci()

for i in range(15):

print(fib(i), end=", ")

The output:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,

Write a function which calculates the arithmetic mean of a variable number of values.

#### Solution

def arithmetic\_mean(x, \*l):

""" The function calculates the arithmetic mean of a non-empty

arbitrary number of numbers """

sum = x

for i in l:

sum += i

return sum / (1.0 + len(l))

def fib(n):

if n == 0:

return 0

elif n == 1:

return 1

else:

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

def ifib(n):

a, b = 0, 1

for i in range(n):

a, b = b, a + b

return a

>>> import re

>>> x = re.search("cat","A cat and a rat can't be friends.")

>>> print x

<\_sre.SRE\_Match object at 0x7fd4bf238238>

>>> x = re.search("cow","A cat and a rat can't be friends.")

>>> print x

None

But what, if we want to match a regular expression at the beginning of a string and only at the beginning?   
The re module of Python provides two functions to match regular expressions. We have met already one of them, i.e. search(). The other has in our opinion a misleading name: match()   
Misleading, because match(re\_str, s) checks for a match of re\_str merely at the beginning of the string.   
But anyway, match() is the solution to our question, as we can see in the following example:

>>> import re

>>> s1 = "Mayer is a very common Name"

>>> s2 = "He is called Meyer but he isn't German."

>>> print re.search(r"M[ae][iy]er", s1)

<\_sre.SRE\_Match object at 0x7fc59c5f26b0>

>>> print re.search(r"M[ae][iy]er", s2)

<\_sre.SRE\_Match object at 0x7fc59c5f26b0>

>>> print re.match(r"M[ae][iy]er", s1)

<\_sre.SRE\_Match object at 0x7fc59c5f26b0>

>>> print re.match(r"M[ae][iy]er", s2)

None

>>>

re.findall(pattern, string[, flags])

findall returns all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order in which they are found.

>>> t="A fat cat doesn't eat oat but a rat eats bats."

>>> mo = re.findall("[force]at", t)

>>> print mo

['fat', 'cat', 'eat', 'oat', 'rat', 'eat']

The general syntax of a lambda function is quite simple:  
lambda argument\_list: expression   
The argument list consists of a comma separated list of arguments and the expression is an arithmetic expression using these arguments. You can assign the function to a variable to give it a name.   
The following example of a lambda function returns the sum of its two arguments:

>>> f = lambda x, y : x + y

>>> f(1,1)

2

The advantage of the lambda operator can be seen when it is used in combination with the map() function.   
map() is a function with two arguments:

r = map(func, seq)

The first argument *func* is the name of a function and the second a sequence (e.g. a list) *seq*. *map()* applies the function *func* to all the elements of the sequence *seq*. It returns a new list with the elements changed by *func*

def fahrenheit(T):

return ((float(9)/5)\*T + 32)

def celsius(T):

return (float(5)/9)\*(T-32)

temp = (36.5, 37, 37.5,39)

F = map(fahrenheit, temp)

C = map(celsius, F)

In the example above we haven't used lambda. By using lambda, we wouldn't have had to define and name the functions fahrenheit() and celsius(). You can see this in the following interactive session:

>>> Celsius = [39.2, 36.5, 37.3, 37.8]

>>> Fahrenheit = map(lambda x: (float(9)/5)\*x + 32, Celsius)

>>> print Fahrenheit

[102.56, 97.700000000000003, 99.140000000000001, 100.03999999999999]

>>> C = map(lambda x: (float(5)/9)\*(x-32), Fahrenheit)

>>> print C

[39.200000000000003, 36.5, 37.300000000000004, 37.799999999999997]

>>>

>>> a = [1,2,3,4]

>>> b = [17,12,11,10]

>>> c = [-1,-4,5,9]

>>> map(lambda x,y:x+y, a,b)

[18, 14, 14, 14]

>>> map(lambda x,y,z:x+y+z, a,b,c)

[17, 10, 19, 23]

>>> map(lambda x,y,z:x+y-z, a,b,c)

[19, 18, 9, 5]

>>> fib = [0,1,1,2,3,5,8,13,21,34,55]

>>> result = filter(lambda x: x % 2, fib)

>>> print result

[1, 1, 3, 5, 13, 21, 55]

>>> result = filter(lambda x: x % 2 == 0, fib)

>>> print result

[0, 2, 8, 34]

>>>

>>> f = lambda a,b: a if (a > b) else b

>>> reduce(f, [47,11,42,102,13])

102

>>>

Calculating the sum of the numbers from 1 to 100:

>>> reduce(lambda x, y: x+y, range(1,101))

5050

In the chapter on lambda and map() we had designed a map() function to convert Celsius values into Fahrenheit and vice versa. It looks like this with list comprehension:

>>> Celsius = [39.2, 36.5, 37.3, 37.8]

>>> Fahrenheit = [ ((float(9)/5)\*x + 32) for x in Celsius ]

>>> print Fahrenheit

[102.56, 97.700000000000003, 99.140000000000001, 100.03999999999999]

>>>

The following list comprehension creates the Pythagorean triples:

>>> [(x,y,z) for x in range(1,30) for y in range(x,30) for z in range(y,30) if x\*\*2 + y\*\*2 == z\*\*2]

[(3, 4, 5), (5, 12, 13), (6, 8, 10), (7, 24, 25), (8, 15, 17), (9, 12, 15), (10, 24, 26), (12, 16, 20), (15, 20, 25), (20, 21, 29)]

>>>

Cross product of two sets:

>>> colours = [ "red", "green", "yellow", "blue" ]

>>> things = [ "house", "car", "tree" ]

>>> coloured\_things = [ (x,y) for x in colours for y in things ]

>>> print coloured\_things

[('red', 'house'), ('red', 'car'), ('red', 'tree'), ('green', 'house'), ('green', 'car'), ('green', 'tree'), ('yellow', 'house'), ('yellow', 'car'), ('yellow', 'tree'), ('blue', 'house'), ('blue', 'car'), ('blue', 'tree')]

>>>

### Generator Compreh

### Generator Comprehension

Generator comprehensions were introduced with Python 2.6. They are simply a generator expression with a parenthesis - round brackets - around it. Otherwise, the syntax and the way of working is like list comprehension, but a generator comprehension returns a generator instead of a list.

>>> x = (x \*\*2 for x in range(20))

>>> print(x)

at 0xb7307aa4>

>>> x = list(x)

>>> print(x)

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361]

Fn = Fn - 1 + Fn - 2   
with the seed values:  
F0 = 0 and F1 = 1

def fibonacci(n):

"""Fibonacci numbers generator, first n"""

a, b, counter = 0, 1, 0

while True:

if (counter > n): return

yield a

a, b = b, a + b

counter += 1

f = fibonacci(5)

for x in f:

print x,

print

#!/usr/bin/python

import MySQLdb

# Open database connection

db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using *cursor()* method

cursor = db.cursor()

# Drop table if it already exist using *execute()* method.

cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# Create table as per requirement

sql = """CREATE TABLE EMPLOYEE (

FIRST\_NAME CHAR(20) NOT NULL,

LAST\_NAME CHAR(20),

AGE INT,

SEX CHAR(1),

INCOME FLOAT )"""

cursor.execute(sql)

# disconnect from server

db.close()

#!/usr/bin/python

import MySQLdb

# Open database connection

db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using *cursor()* method

cursor = db.cursor()

# Prepare SQL query to INSERT a record into the database.

sql = "INSERT INTO EMPLOYEE(FIRST\_NAME, \

LAST\_NAME, AGE, SEX, INCOME) \

VALUES ('%s', '%s', '%d', '%c', '%d' )" % \

('Mac', 'Mohan', 20, 'M', 2000)

try:

# Execute the SQL command

cursor.execute(sql)

# Commit your changes in the database

db.commit()

except:

# Rollback in case there is any error

db.rollback()

# disconnect from server

db.close()

import os

import sys

from sqlalchemy import Column, ForeignKey, Integer, String

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import relationship

from sqlalchemy import create\_engine

Base = declarative\_base()

class Person(Base):

    \_\_tablename\_\_ = 'person'

    # Here we define columns for the table person

    # Notice that each column is also a normal Python instance attribute.

    id = Column(Integer, primary\_key=True)

    name = Column(String(250), nullable=False)

class Address(Base):

    \_\_tablename\_\_ = 'address'

    # Here we define columns for the table address.

    # Notice that each column is also a normal Python instance attribute.

    id = Column(Integer, primary\_key=True)

    street\_name = Column(String(250))

    street\_number = Column(String(250))

    post\_code = Column(String(250), nullable=False)

    person\_id = Column(Integer, ForeignKey('person.id'))

    person = relationship(Person)

# Create an engine that stores data in the local directory's

# sqlalchemy\_example.db file.

engine = create\_engine('sqlite:///sqlalchemy\_example.db')

# Create all tables in the engine. This is equivalent to "Create Table"

# statements in raw SQL.

Base.metadata.create\_all(engine)

Fibonacci using recursion:

def recur\_fibo(n):

"""Recursive function to

print Fibonacci sequence"""

if n <= 1:

return n

else:

return(recur\_fibo(n-1) + recur\_fibo(n-2))

# Change this value for a different result

nterms = 10

# uncomment to take input from the user

#nterms = int(input("How many terms? "))

# check if the number of terms is valid

if nterms <= 0:

print("Plese enter a positive integer")

else:

print("Fibonacci sequence:")

for i in range(nterms):

print(recur\_fibo(i))

# Python Program to display the powers of 2 using anonymous function

# Change this value for a different result

terms = 10

# Uncomment to take number of terms from user

#terms = int(input("How many terms? "))

# use anonymous function

result = list(map(lambda x: 2 \*\* x, range(terms)))

# display the result

print("The total terms is:",terms)

for i in range(terms):

print("2 raised to power",i,"is",result[i])

def binary\_sort(sortedlist,n,x):

start = 0

end = n - 1

while(start <= end):

mid = (start + end)/2

if (x == sortedlist[mid]):

return mid

elif(x < sortedlist[mid]):

end = mid - 1

else:

start = mid + 1

return -1

n = input("Enter the size of the list: ")

sortedlist = []

for i in range(n):

sortedlist.append(input("Enter %dth element: "%i))

x = input("Enter the number to search: ")

position = binary\_sort(sortedlist, n, x)

if(position != -1):

print("Entered number %d is present at position: %d"%(x,position))

else:

print("Entered number %d is not present in the list"%x)

Fibonnaci recursion efficient:

memo = {0:0, 1:1}

def fibm(n):

if not n in memo:

memo[n] = fibm(n-1) + fibm(n-2)

return memo[n]

remembering the calculated value