List are fun.

Python knows a number of compound data types, which are used to group different data types. The most versatile is the list. To create a list we first define a variable, that holds the list, and assign this to a list. The list it self is contained in square brackets and the items (values) are separated by commas like this:

```
my_list = [1, 23, 'abc', 45.6, 2555, 'lists are fun']
```

You can create a new empty list in two ways:

```
>>> new_list = []
>>> new_list
[]
>>> my_list = list()
>>> my_list
[]
```

Visually, the first way of defining a new empty list are to be preferred, since the square brackets indicates that we have a list and nothing in inserted to the list. The second way of defining a new empty list is even as valid, but here it more indicates, that the definition of Python list originates from a class function. At this point we will not discuss classes, but just use these as they are.

Lists are mutable, i.e. they can be changed at any time during a runtime process. This means, that values can be changed, removed and added.

Like in strings we can access each element in a list by indices. The first element in a list have the indices = 0, second element indices = 1 and so on. This gives us the option to access any element in a list.
As seen to the right, we easily can extract the values from the my_list using indices. Note that my_list[-1] returns the last element in the list. This indicates, that the indices in a list "runs" in a modulo fashion way that are dependent on the length of the list. However, the last element of a list can always be found independent of the length of the list by using the indices -1. It is worth mentioning, that if the indices used to retrieve a value, is larger than the indices registered in the list, Python raises an IndexError.

```
>>> my_list = [1, 2, 3, 4, 5, 6, 'a word']
>>> my_list[0]
1
>>> my_list[4]
5
>>> my_list[-1]
'a word'
>>> my_list[6]
'a word'
>>> my_list[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

```
>>> my_list[:3]
[1, 2, 3]
>>> my_list[3:]
[4, 5, 6, 'a word']
>>> my_list[6:]
['a word']
>>> my_list[3: 6]
[4, 5, 6]
```

Aside from accessing single elements in a list, we can extract multiple values by slicing the list. When we slice a list we use the indices to determine where in the list we want to slice this. To slice a list we use the following notation:
listname[start_index, end_index], the end_index are not included. Note, that my_list[:3] does not need the start index set to 0 as this is done by default. Likewise, the notation my_list[3:] will extract all values from indices 3 to the end of the list.

Mutability

Lists are mutable, i.e. we can change the values and content of a list. This makes Python list very powerful, but also very fragile at the same time. In other words, we can unintended make changes to a list if several functions in a code have access to it.

The two examples here explains how mutability works.

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a
>>> c += b
>>> a
[1, 2, 3, 4, 5, 6]
>>> b
[4, 5, 6]
>>> c
[1, 2, 3, 4, 5, 6]
```

```
>>> lst_a = [1, 2, 3]
>>> lst_b = lst_a
>>> lst_b[0] = 6
>>> lst_a
[6, 2, 3]
>>> lst_b
[6, 2, 3]
```

Some methods.

We have several different methods which we can use on lists. Among those are: append, remove, count, pop, insert, sort and some few others. These methods offers a possibility to manipulate the content of the list. If we want to add a new item to our list we would use the append method. This method appends an item as the last element in the list. The pop() method by default removes the last item in a list and returns this.

```
>>> a = [1, 2, 3]
>>> a.append(5)
>>> a
[1, 2, 3, 5]
>>> a.append(6)
>>> a
[1, 2, 3, 5, 6]
>>> a.pop()
6
>>> a.remove(1)
>>> a
[2, 3, 5]
```

Lists used as stacks.

You can use lists as a stack. This means, that whatever you put into the list will be the first element you take out. When we do this we use the above methods append and pop. The pop method returns the value and removes the last entry in the list. This procedure are also known as “Last In First Out” or LIFO.

List as queues.

You can also use list as queues. Here we take the first element in the list and remove it. This procedure are also know as “First In First Out” or FIFO. When we use a list as queues we do face a problem. Every time we remove the first entry, the remaining items in the list have to change places, i.e. one shift to the left. This means, that when we are handling very large lists as queues, this procedure is slow and memory consuming. Python can deal with this and the collections module contains the deque method, which is designed to have fast appends and pops from both ends of a list.

```
>>> my_list = [1, 2, 3, 4, 5, 6]
>>> my_list.pop(0)
1
>>> my_list.pop(0)
2
>>> my_list
[3, 4, 5, 6]
```

```
>>> from collections import deque
>>> my_list = deque([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> my_list.append(10)
>>> my_list.append(11)
>>> my_list
deque([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
>>> my_list.popleft()
1
>>> my_list.popleft()
2
>>> my_list[0]
3
```

List comprehensions.

Ever wondered about the degree of despair you eventually could reach if you had to handle a mixed list containing strings, floats and integers, and you were asked to separate the list into 3 list, respectively holding each type apart from each other?

Well, you are at the moment trying to solve this “tiny” problem and all you see in the dim light of a solution is loads of iterating, appending and possible errors.

Python can do it the easy way and you don't even have to wrinkle your grey cells.

So consider this code:

```
>>> mixed = [0, 1, 2, 3, 'a', 4, 5, 6.0, 'b', 'c', 7, 8, 'd', 9]
```

We can easily determine the length of the mixed list.

```
>>> len(mixed)
14
```

But what if I asked you about the numbers of integers and floats in mixed? Are you already thinking of large code? Don't , there's an easy way – almost too easy – we use list comprehensions.

```
>>> len([x for x in mixed if type(x) != str])
10
```

That was easy. The above code are similar to the code below.

```
>>> count = 0
>>> for n in mixed:
...     if type(n) != str:
...         count += 1
...
>>> count
10
```

What about the sum of all the numbers in mixed?

```
>>> sum([x for x in mixed if type(x) != str])
45.0
```

At this point you've maybe already guessed the solution to separate the list into 3 different lists: One with strings, one with integers and one with floats. So here we go.

```
>>> strings = [x for x in mixed if type(x) == str]
>>> floats = [x for x in mixed if type(x) == float]
>>> integers = [x for x in mixed if type(x) == int]
>>> int_and_floats = [x for x in mixed if type(x) != str]
>>> strings
['a', 'b', 'c', 'd']
>>> floats
[6.0]
>>> integers
[0, 1, 2, 3, 4, 5, 7, 8, 9]
>>> int_and_floats
[0, 1, 2, 3, 4, 5, 6.0, 7, 8, 9]
```

You can read more about list comprehensions in the Python documantation.

List are fun – part II.

List is, with tuples and libraries, the backbone of Python. In this part we'll see how we can mutate a list by using a function and apply the function to the values in the list. But first a word on how to write your functions.

Basic concepts on writing functions.

Examples from the file function_on_list.py. You can download it from our Files Tab.

Writing a function square(x) like this:

```
def square(x):
    return x ** 2
```

doesn't tell the user very much. Applying a doc string gives the user the needed information on how the function works:

```
def square(x):
    '''(number) -> number
    Returns the square of a given
number x.
    >>> square(3)
    9
    >>> square(-2)
    4
    >>> square(0)
    0
    '''
    return x ** 2
```

In the doc string there's a type contract: (number) → number. In this case, the function takes ints and floats as parameters. Then comes the description: Returns the square of a given number x. This part of the doc string tells the user what the function actually do. The rest of the doc string is

referred to as a test suite. Test suites are important. Here you test the function with appropriate values of special interest, i.e. extreme low values, values close to zero, extreme high values or what values may be of special interest depending on how the function are constructed. E.g. when working with strings it may be of interest to evaluate whether the function only works strings of equally length.
The advantage in using the doc strings are, that when you write the functions name in IDLE the doc string automatically appears as a help to use the function.

Back to list.

In the function_on_list.py i've defined 3 functions square, add and root. These functions can be used on the 2 different list lst_a or lst_b. To do that I've created a function called list_function(list, function). The list_function mutates, i.e. changes the values in the list, and prints the new values in the list. E.g. list_function(lst_a, add) prints the new list lst_a = [2, 4, 6, 8]. You can define any list, i.e. list_function([11, 13, 15, 16, 18, 20], add) and still apply the chosen function to the list.
You can also apply built-in functions to the list. Consider we have a list of mixed numbers lst = [-4, -5 , 2, 34, 2, -7, 8] we can easily convert all numbers in the list to positive numbers as follows: list_function([-4, -5 , 2, 34, 2, -7, 8], abs) returns [4, 5, 2, 34, 2, 7, 8].

A step further on list and functions.

The function choose() lets the user choose between 3 different function that can be applied to the lst_a. Depending of the users input the choice is executed end returned. In this example we see, that it's possible to store the functions in a list func_list, then find the specified function and execute the choice. A slight modification to the function choose can make it work in a game, i.e. if an external value (raw_input) reach a certain value it, could change the speed of an object or other conditions in the game.

The root function. - avoiding errors.

It's almost an art to debug code and there are less difficult ways to deal with errors. One is exceptions. In the root function we all know that we can't use negative numbers (not yet), so a negative number in a list will eventually raise an error. To handle this error we use the try and except. This doesn't change our function, but it tells Python first to try to return the value x ** 0.5. If the value can't be returned there is an error. In case of negative numbers a ValueError. So when the ValueError is raised we have made an exception that prints a message to the user and ends with a finally remark.

The easy way to do the stuff.

In the above I've made my own function which applied a function to a given list. But there are easier ways to handle this. In Python there's a built-in function called map(function, listA, listB).

The map function works like in the examples shown to the right.

The map(min, lst_a, lst_b) takes the corresponding values in lst_f and lst_g and returns the lowest value, i.e. in lst_f the value 2 is larger than the corresponding value 1 in lst_g.

```
>>> lst_f = [2, 4, 6, 8]
>>> lst_g = [1, 5, 7, 9]
>>> map(min, lst_f, lst_g)
[1, 4, 6, 8]
>>> map(max, lst_f, lst_g)
[2, 5, 7, 9]
>>> map(square, lst_f)
[4, 16, 36, 64]
```

Remember Python is fun.
Holmes.