1 Python Scripting für Physiker - Handout W1

2 Built-In Types

2.1 Numbers

Number data types store numeric values (e.g integer,long,float,complex).

```
>>> var1 = 4          # assign integer 4 to variable var1
>>> var2 = 4.         # assign float 4.0 to variable var2
>>> var1,var2 = 3,5 # assign multiple variables at once
```

They work with common operators +,-,*,/.
(see 5.4 *http : //docs.python.org/library/stdtypes.html* for a complete list of operators)

```
>>> a = 6/5           # assigns integer result of 6/5 (1) to variable a
>>> b = 6./5.         # assigns float result of 6/5 (1.2) to variable b
```

2.2 Strings

Strings in Python are identified as a contiguous set of characters in between quotation marks.

```
>>> a = "Spam_Eggs"      # assigns sequence of characters to variable a
>>> a = 'Spam_Eggs'      # does the same
```

Strings also work with common operators +,*.

```
>>> a = "Spam_Eggs"      # assigns sequence of characters to variable a
>>> a * 2
...'Spam_EggsSpam_Eggs'
>>> a + 'Eggs'
...'Spam_EggsEggs'
```

There are also several string methods available, that may help working with strings. For now, you may think of methods as functions attached to an object (in this case a string object), which are triggered with a call expression and may or may not return some result.

```
Syntax:
result = var.method_name(args)
```

Methods are covered in more detail later on.

```
>>> a = "SpamEggs"
>>> result = a.upper()    # transform string into uppercase
>>> print result
...'SPAMEGGS'
>>> result = a.replace('Spam',"Boiled") # replace substring with new one
>>> print result
...'BoiledEggs'
```

2.3 Lists

Lists are the most versatile of Python's data types. A list contains items separated by commas and enclosed within square brackets ([]). They work with operators +,* and slices just like strings.

```
>>> a = [1,4,'Spam','Eggs']     # Assigns a list to variable a
>>> a + [4]                     # Operators work like with strings
...[1,4,'Spam','Eggs',4]
>>> a * 2
...[1,4,'Spam','Eggs',1,4,'Spam','Eggs']
```

Also a whole bunch of methods exist for lists.

```
>>> a.append('Beans')            # Adds an additional item to the list
>>> print a
...[1,4,'Spam','Eggs','Beans']
>>> a.remove('Eggs')             # Removes the first entry containing 'Eggs' in the list
>>> print a
...[1,4,'Spam','Beans']          # Notice how 'Eggs' is now missing,
                                 # but 'Beans' is still in there!
```

A more extended list of methods is found at
http : *//docs.python.org/tutorial/datastructures.html*.

3 Getting Help

The methods introduced in the prior section are a representative, but small, sample of what is available. For a complete list of an objects methods you can call the the built-in *dir* function.

```
>>> dir("aString")
...[...,'replace',...,'upper',...]
```

To ask what they do, you can call the *help* function.

```
>>> help("aString".upper)
...Help on built-in function upper:
...
...upper(...)
...     S.upper() -> string
...
...     Return a copy of the string S converted to uppercase.
...(END)
```

4 Indexing and Slicing

Strings and Lists are sequences - a positionally ordered collection of other object. Sequences maintain a left-to-right order among the items they contain. Their items are stored and fetched by their relative position. Strings for example are sequences of one-character strings.
To fetch the objects inside sequences we use Python's indexing expressions

4.1 Indexing

```
With strings
>>> a = "Spam_Eggs"
   +---+---+---+---+---+---+---+---+---+
   | S | p | a | m | _ | E | g | g | s |
   +---+---+---+---+---+---+---+---+---+
     0   1   2   3   4   5   6   7   8
    -9  -8  -7  -6  -5  -4  -3  -2  -1
>>> a[0]    # The first item in a
...'S'
>>> a[1]    # The second item in a
...'p'
>>> a[-1]   # The last item in a
...'s'
>>> a[-2]   # The second last item in a
...'g'
Indexing techniques work the same way with lists
>>> k = ["Spam","_","Eggs"]
>>> k[0]
...'Spam'
>>> k[-1]
...'Eggs'
And even on multiple layers
>>> k[-1][0]
...'E'
```

4.2 Slicing

In addition to simple positional indexing, sequences also support a more general form of indexing known as slicing. Slicing allows to extract an entire section (slice) in a single step.

```
With Strings
>>> a = "Spam_Eggs"
>>> a[0:4]   # Slice of a from index 0 through 3 (not 4)
...'Spam'
>>> a[5:]    # Slice of a from index 5 to the end
...'Eggs'
>>> a[1:6:2] # The third parameter is the increment -> indices 1,1+2,1+4 are returned
...'pmE'
>>> a[5::-1] # Slice of a from index 5 to 0 (starting at index 5 with an increment of -1)
 ...'E_mapS'
Again the same for lists
>>> k = [0,1,2,3,4]
>>> k[0:3:2]
...[0,2]
```

5 Control Structures

5.1 Boolean Expressions

Evaluating Boolean expressions and continue based on its value beeing 'True' or 'False' is a fundamental concept of (Python-) programming.

```
>>> a = 5.
>>> b = 6.
Syntax:
>>> a == b            # checks whether a equals b or not (Note the difference between '==' and '=')
...'False'
>>> a != b            # checks whether a does not equals b or not
...'True'
>>> a >= b            # checks whether a greater or equal b or not
...'False'
>>> a < b             # checks whether a lighter b or not
...'True'
Simple Boolean expression can be connected by Boolean operators and,or,not
>>> c = 4.
>>> a < b and b < c
...'False'
>>> a < b and not b < c
...'True'
```

5.2 If, Else, Elif Statement

The if statement of Python is similar to that of other languages. The if statement evaluates the result of a Boolean expression and continues based on the result beeing True or False.

```
Syntax:
                    # Note: Python uses indentation as its method of grouping statements
if expression1:     # If expression1 is True:
   statement1       # execute statement1
elif expression2:   # If expression1 is not True and expression2 is True:
   statement2       # execute statement2
...
elif expressionN-1:
   statementN-1
else:               # If expression1, expression2, ... expressionN-1 are not True:
   statementN       # execute statementN
```

Here is a very simple example using the if-statement.

```
>>> if 3==4:
>>>     k = 'Spam'
>>>     print k
>>> elif 3 <= 4:
>>>     print 'Eggs'
>>> else:
>>>     print 'Spam_Eggs'
...'Eggs'
```

5.3 While, For Loops

A loop is a construct that causes a section of a program to be repeated a certain number of times. The repetition continues while the condition set for the loop remains true. When the condition becomes false, the loop ends and the program control is passed to the statement following the loop.

5.4 While

The while loop is one of the looping constructs available in Python. The while loop continues until the expression becomes false. The expression has to be a logical expression and must return either a true or a false value

```
Syntax:

while expression == True:     # while expression is True:
   statement                  # execute statement
```

Here is a very simple example using the while loop.

```
>>> k = 0
>>> while k<3:
>>>     print 'k = ',k
>>>     k = k+1
...k = 0
...k = 1
...k = 2
```

5.5 For

The for loop in Python has the ability to iterate over the items of any sequence, such as a list or a string.

```
Syntax:

for object in sequence:     # For every object in sequence:
   statement                # execute statement
```

Here is a very simple example using the for loop.

```
k = [0,1,2]
>>> for item in k:   # Note: You can name the iterator variable (here 'item') as you like
>>>     print 'item = 'item
...item = 0
...item = 1
...item = 2
```

6 Exercise

Mandatory Function to solve the excercise

```
>>> aList = range(n,N) # Creates a list with numbers from n to N-1
>>> ListLength = len(aList) # Returns the length of a given sequence
```

1. Create a list containing all integers from 0 to 100. Now create a second list containing the squares of this integers.
2. Now create a third list only containing the odd entries of the second list.
3. (Advanced) Calculate all prime numbers up to N = 100.