Use of Python functions

```
      #    We used some simple builtin functions before, like:
F1001 >>>  min(9, 4, 6, 8)   # return the smallest value from a list
      ==>  4
F1002 >>>  drinks = ["whiskey", "beer", "wine", "juice", "water"]  # define a list
F1003 >>>  sorted(drinks)  # returns a new list
      ==>  ['beer', 'juice', 'water', 'whiskey', 'wine']
F1004 >>>  print("hello world")
      p()  hello world
F1005 >>>  round(5.53703, 2)
      ==>  5.54
F1006 >>>  bool(2+4==6)
      ==>  True
```

Create our own function

```
        #    let's create a function with the name 'foo'. Does it already exist?
F1007 >>>    foo
      err!   NameError("name 'foo' is not defined",)
        #    to create our own functions, we use the 'def' statement (=define)
F1008 >>>    def foo():
      ...        print('this is the foo-function')
        #    the code inside the function is indented one level
        #    the function is now defined, python knows its name:
F1009 >>>    foo
      ==>    <function foo at 0x00000000034946A8>
        #    now we call the function - using the name with parenthesis
F1010 >>>    foo()
      p()    this is the foo-function
        #    we see: calling a function means to execute the statements inside
```

Functions can take arguments

```
F1011 >>>   def foo(text):
      ...       print("foo was called with '{}'".format(text))
        #   we just used the same name for new function. Now call it:
F1012 >>>   foo('stars')
      p()   foo was called with 'stars'

        #   several arguments can be used:
F1013 >>>   def foo(arg1, arg2):
      ...       print("called with arguments: {}, {}".format(arg1, arg2))
F1014 >>>   foo(987, 38)
      p()   called with arguments: 987, 38
```

Functions can return results

```
        #    calculate and return a value
F1015 >>>    def square(arg1):
      ...        prod = arg1 * arg1
      ...        return prod
F1016 >>>    square(16)
      ==>    256

F1017 >>>    def multiply(fact1, fact2):
      ...        return fact1 * fact2
F1018 >>>    multiply(7, 13)
      ==>    91

        #    try a special function:
F1019 >>>    def python_is_cool():
      ...        return True
F1020 >>>    python_is_cool()  # no further argument is needed :-)
      ==>    True
```

Return multiple values

```
F1021 >>>   drinks = ["whiskey", "beer", "wine", "juice", "water"]  # define a list

F1022 >>>   def first_and_last(sequ):  # take a sequence (like a list)
      ...       first = sequ[0]    # get the first element
      ...       last = sequ[-1]    # and the last
      ...       return first, last  # return more than one value
F1023 >>>   first_and_last(drinks)  # this returns a tuple
      ==>   ('whiskey', 'water')

F1024 >>>   df, dl = first_and_last(drinks)  # 'unpack' the tuple
F1025 >>>   dl
      ==>   'water'

F1026 >>>   both = first_and_last("LISBOA")  # yes, a string can be used like a list
F1027 >>>   both    # the tuple was assigned to a variable
      ==>   ('L', 'A')

        #   In the Collection-OOTS there is a section about tuples,
        #   which explains the handling of tuples more detailed
```

Return from a function

```
        #    A function does not need a return statement
F1028 >>>    def noreturn(text):
      ...        print("called with:", text)
F1029 >>>    result = noreturn('some text')
      p()    called with: some text
F1030 >>>    str(result)  # Show the 'None' result
      ==>    'None'

        #    A function can have more than  one return statement
F1031 >>>    def decide(p1, p2):
      ...        if p2 == True:
      ...            return p1
      ...        print('no', p1)
      ...        return False
F1032 >>>    decide("success", True)
      ==>    'success'
F1033 >>>    decide("success", False)
      p()    no success
      ==>    False
```

Function arguments with defaults (1)

```
      #    An argument can have a default (predefined) value
F1034 >>>  def decide(p1, p2=False):
      ...      if p2 == True:
      ...          return p1
      ...      print('no', p1)
      ...      return False
F1035 >>>  decide("success")  # argument p2 is not specified on call
      p()  no success
      ==>  False
F1036 >>>  decide("success", True)
      ==>  'success'
```

Function arguments with defaults (2)

```
        #    Arguments without a default value must be specified at the function call
F1037 >>>    def manyargs(name, number, third="3rd", fourth=None, fifth="5th"):
      ...        print("name:{}, number:{}, third:{}, fourth:{}, fifth:{}"
      ...              .format(name, number, third, fourth, fifth))
F1038 >>>    manyargs('Tom', 17)
      p()    name:Tom, number:17, third:3rd, fourth:None, fifth:5th
F1039 >>>    manyargs('Tom', 17, 'drei')  # specify arguments by position
      p()    name:Tom, number:17, third:drei, fourth:None, fifth:5th
F1040 >>>    manyargs('Tom', 17, fourth='vier')  # specifiy an argument by name
      p()    name:Tom, number:17, third:3rd, fourth:vier, fifth:5th
F1041 >>>    manyargs('Tom', number=17, fifth='fünf')  # specify by name also for args without default
      p()    name:Tom, number:17, third:3rd, fourth:None, fifth:fünf

        #    On a function definition:
        #        First arguments without defaults, then arguments with defaults
        #    On a function call:
        #        First arguments without names (positional) then args with names (keyword)
```

Functions used for a general program structure

```
#    Most of the code of a program should be enclosed in functions,
#    starting with a 'main()' function, followed by other function definitions
#    The last statement in the program is then the call of the main function.

F1042 >>>   # python3  #
...         """ This is a sample for the general form
...             of a python program
...         """
...         import random  # import statements
...
...         def main():
...             print("start of main function")
...             result = sample('test')
...             print("result:", result)
...
...         def sample(arg1):
...             print("start of sample function")
...             return arg1 +' '+ str(random.random() * 999)
...
...         print("call of the main function")
...         main()
p()         call of the main function
p()         start of main function
p()         start of sample function
p()         result: test 182.6217551981555
```

Functions as objects

```
        #    Everything in Python is an object, ==> functions are objects

F1043 >>>    def foo():     # the def creates a name (foo) and connects it to a function object
      ...        print('this is foo()')
      ...        return True
F1044 >>>    foo    # show the name of a function
      ==>    <function foo at 0x0000000003499488>
F1045 >>>    foo()  # execute the function
      p()    this is foo()
      ==>    True
F1046 >>>    other = foo   # a new name for the same object
F1047 >>>    other()  # can be used to call the function as well - wow!
      p()    this is foo()
      ==>    True
```

Functions as arguments

```
F1048 >>>   limit = 4
F1049 >>>   def check_limit(arg1):
      ...       if arg1 > limit:  # we can access a variable, which is defined 'outside'
      ...           print("{} is too big".format(arg1))

F1050 >>>   check_limit(3)
F1051 >>>   check_limit(8)  # to call the function works as expected
      p()   8 is too big

        #   but what goes on here:
F1052 >>>   def repeater(count, func):
      ...       limit = 6  # this does not overwrite the 'limit' defined outside
      ...       for val in range(count):
      ...           func(val)  # what does happen here?

F1053 >>>   repeater(8, check_limit)
      p()   5 is too big
      p()   6 is too big
      p()   7 is too big

        #   We call the repeater() with a function object, which is then executed inside.
        #   The repeater can use any function, that is called with one numeric argument.
        #   check_limit() is defined locally and is later called from 'outside'. A function, handed
        #   over to someone else, who calls the function later, is a 'call-back' function.
```

The "Scope"

```
        #   The term 'scope' is about the places, where a variable is visible and can be changed.
        #   Functions have their own scope - variables defined inside a function are local,
        #   they are not visible from outside.

F1054 >>>   def myfunc(arg1):
      ...       oldname = name
      ...       name = 'Thomas'
      ...       myarg = arg1
      ...       return oldname + myarg + name

F1055 >>>   name = 'Joan'
F1056 >>>   arg2 = myfunc(' -- ')
F1057 >>>   name, arg2
      ==>   ('Joan', 'Joan -- Thomas')
F1058 >>>   arg1    # existed only inside the function
      err!  NameError("name 'arg1' is not defined",)
F1059 >>>   arg2    # this was returned from the function
      ==>   'Joan -- Thomas'

        #   When a function is executed, there are two scopes: the local scope inside the function
        #   and the global scope (the 'module' scope).
        #   When a variable is 'used' (read), the name 'lookup' is first in the local,
        #   then in the global scope
        #   When a variable is created (write) the name is always created in the local scope.
        #   The local name hides the (same) global name
```

The Scope

```
#   For classes/objects there is a third form of scope:
#   ==> the object level, represented by 'self'
#   A method in an class has a local scope, names stored here destroyed after the method returns
#   Any data that lives longer than the method, is stored in self
#   Class methods never access global data

#   The data in self is visible (and can even be changed) from outside
#   Object attributes (properties) "can" be read, but must never be changed directly
#   Objects should provide methods, which make a direct access from outside unnecessary
```

Recursion: Divide and Conquer

```
        #    A function can call itself
        #    But before we start with recursion, we must understand, what recursion is!

        #
F1060 >>>    eval('2+3')   # first show the 'eval()' function - interpret a string as a Python expression
      ==>    5
F1061 >>>    eval('2+x')   # which may of course fail
      err!   NameError("name 'x' is not defined",)
        #    let's assume, eval() could not work with parenthesis (which it can of course)

F1062 >>>    xpress = '1+2+(3*(4+7)+8/(2*2))*6+(3+7*(8-6))'   # how to calculate this value
        #    As humans we calculate the innermost expressions first, eliminating the
        #    nested parenthesis pair by pair. But we want to hand this task over to a function.

        #    How would a function do this?
```

A recursive solution ... let a clone do your work

```
F1063 >>>   def recurs(text, lvl=0):
      ...       evalstr = ''
      ...       while text:
      ...           p = text[0]
      ...           text = text[1:] if text != p else ''
      ...           if p == '(':
      ...               val, text = recurs(text, lvl+1)  # here it happens!
      ...               evalstr += str(val)
      ...           elif p == ')':
      ...               break
      ...           else:
      ...               evalstr += p
      ...       # evaluate, what was inside the () and return the rest of the text
      ...       print("level {} evaluate: {}".format(lvl, evalstr))
      ...       result = eval(evalstr) if evalstr else ''
      ...       return result, text  # eval() never gets any parenthesis

        #   ... and how it works:
F1064 >>>   recurs('3+4*5')  # start very simple
      p()   level 0 evaluate: 3+4*5
      ==>   (23, '')
F1065 >>>   recurs('3+(4*5)/2')  # one level deeper
      p()   level 1 evaluate: 4*5
      p()   level 0 evaluate: 3+20/2
      ==>   (13.0, '')
```

Recursive evaluation at work

```
F1066 >>>   recurs('(3+3)*3+(4*5)/2*(4+4)')
      p()   level 1 evaluate: 3+3
      p()   level 1 evaluate: 4*5
      p()   level 1 evaluate: 4+4
      p()   level 0 evaluate: 6*3+20/2*8
      ==>   (98.0, '')
F1067 >>>   xpress = '1+2+(3*(4+7)+8/(2*2))*6+(3+7*(8-6))'
F1068 >>>   recurs(xpress)
      p()   level 2 evaluate: 4+7
      p()   level 2 evaluate: 2*2
      p()   level 1 evaluate: 3*11+8/4
      p()   level 2 evaluate: 8-6
      p()   level 1 evaluate: 3+7*2
      p()   level 0 evaluate: 1+2+35.0*6+17
      ==>   (230.0, '')
F1069 >>>   eval(xpress)  # compare against this
      ==>   230.0

F1070 >>>   recurs('()')   # special cases work
      p()   level 1 evaluate:
      p()   level 0 evaluate:
      ==>   ('', '')
F1071 >>>   recurs('(((0)))')
      p()   level 3 evaluate: 0
      p()   level 2 evaluate: 0
      p()   level 1 evaluate: 0
      p()   level 0 evaluate: 0
      ==>   (0, '')

        #   the meditation starts now!
```

Generators - Give me more!

```
        #   Functions always return one value - which may be None or actually a tuple of values.
        #   Functions have zero or more return statements (plus the invisible return at the end).
        #   As soon as one 'return' is reached, the function is terminated, all local names disappear.

        #   There is a statement similar to 'return' --> 'yield' which has a similar meaning.
        #   When a 'yield' statement is reached, the function does not stop, but is paused
        #   when 'called again', the function continues after the yield, with the same internal status.

        #   a first example:
F1072 >>>   def many_numbers(start=0, end=10, step=1):
      ...       i = start
      ...       while i < end:
      ...           yield i
      ...           i += step

F1073 >>>   type(many_numbers)    # it looks like a function, so let's call it:
      ==>   <class 'function'>
F1074 >>>   many_numbers()    # looks like a function call, but returns what?
      ==>   <generator object many_numbers at 0x000000000349C360>
```

Generator - a function returns a generator (object)

```
        #   A Function, which contains a yield statement is changed into a 'Generator'
        #   calling a Generator function returns a "generator object",
        #   which is a special type of an "iterator"

        #   it can be used in a 'for' loop:
F1075 >>>   for num in many_numbers(5,8):   # remember: the function must be called to get the iterator
      ...       print("got number", num)
      p()   got number 5
      p()   got number 6
      p()   got number 7
        #   or in other places, where an iterator is expected:
F1076 >>>   tuple(many_numbers(5,8))    # create tuple
      ==>   (5, 6, 7)
F1077 >>>   [num*3 for num in many_numbers()]  # part of a list comprehension
      ==>   [0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

Get single values from a generator

```
        #    the above examples consume the output of the generator in a single step
        #    there is a special function 'next()', which extracts only one value from the next 'yield'
F1078 >>>    num_gen = many_numbers(3,6)   # first we have to get the iterator
F1079 >>>    next(num_gen)    # now next() will extract the first value
      ==>    3
F1080 >>>    next(num_gen)    # and the second
      ==>    4
F1081 >>>    next(num_gen)    # and the third (=the last value from the loop)
      ==>    5
F1082 >>>    next(num_gen)    # next() tries to go to the next 'yield', but the loop ends and 'returns'
      err!   StopIteration()
        #    the StopIteration exception is not an error, but the 'normal' end of an iterator

        #    Let's try it again:
F1083 >>>    num_gen = many_numbers(3,6)
F1084 >>>    while True:
      ...        try:
      ...            value = next(num_gen)
      ...        except StopIteration:
      ...            break
      ...        else:
      ...            print("received", value)
      ...    print("iteration ended")
      p()    received 3
      p()    received 4
      p()    received 5
      p()    iteration ended
```

Generators can last forever

```
        #    Let's write a generator, which never stops
F1085 >>>    import random
F1086 >>>    def dice():
       ...       sides = (1,2,3,4,5,6)
       ...       while True:
       ...           yield random.choice(sides)
F1087 >>>    cast = dice()
        #    now simulate the throwing of 2 dice
F1088 >>>    next(cast), next(cast)
       ==>   (5, 6)
F1089 >>>    next(cast), next(cast)
       ==>   (5, 1)
F1090 >>>    next(cast), next(cast)
       ==>   (6, 3)
        #    .... this can happen ever again
```

mooooooorre

```
        #    An iterato
        #    An 'iterator'?  - we used iterators before to create lists and tuples ...
F1091 >>>    list(many_numbers())   # The iterator can be used directly
      ==>    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
        #    The iterator can also be assigned to a variable
F1092 >>>    iter_many = many_numbers(5, 8)
        #    an iterator supports one special method:
F1093 >>>    next(iter_many)   # start the loop in the many_numbers() function and stops after the yield
      ==>    5
F1094 >>>    next(iter_many)   # continue the loop, increment the 'i' and stop after the yield again
      ==>    6
F1095 >>>    next(iter_many)   # one more yield
      ==>    7
        #    what will happen next?
F1096 >>>    next(iter_many)   # the loop in side the function is exhausted. At the end there is a 'return'
      err!   StopIteration()
        #    which leads to a special exception

F1097 >>>    iter(many_numbers())
      ==>    <generator object many_numbers at 0x000000000349C750>
```

moore

```
        #    before we can understand Generators (and how beautiful they are...)
        #    we need to learn the iterator interface
        #    we could use the 'many-numbers' example or something we already know:
F1098 >>>    numbers = many_numbers(0, 4)
F1099 >>>    next(numbers)
      ==>    0
F1100 >>>    next(numbers)
      ==>    1
F1101 >>>    numbers.i
      err!   AttributeError("'generator' object has no attribute 'i'",)
F1102 >>>    next(numbers)
      ==>    2
F1103 >>>    numbers
      ==>    <generator object many_numbers at 0x000000000349C6C0>
F1104 >>>    type(numbers)
      ==>    <class 'generator'>
```

mooooooore

```
F1105 >>>   iter((1,2,3))
      ==>   <tuple_iterator object at 0x000000000349F358>
F1106 >>>   iter('hallo welt')
      ==>   <str_iterator object at 0x000000000349F3C8>
F1107 >>>   letters = iter('hallo')
F1108 >>>   next(letters)
      ==>   'h'
F1109 >>>   next(letters)
      ==>   'a'
F1110 >>>   next(letters)
      ==>   'l'
F1111 >>>   next(letters)
      ==>   'l'
F1112 >>>   next(letters)
      ==>   'o'
F1113 >>>   next(letters)
      err!  StopIteration()
```