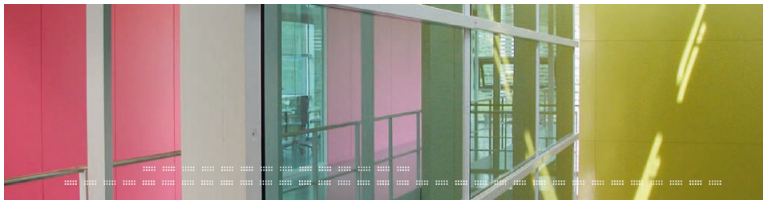


Introduction to Python III

Dipl.-Inf. Domenic Jenz (jenz@hls.de)
HLRS, University of Stuttgart

13. January 2010



Outline

Introduction

Repetition

Sorting again

- Insertion Sort

- Selection Sort

Functions

- Introduction

- Recursive functions

Introduction

My Homepage: www.hlrs.de/people/jenz



Figure: Monty Python. Photo from BBC

Overview

Last year

- ▶ simple calculations in python
- ▶ assignment of values to **variables**
- ▶ different variable types (integer, float, string, list, tuples, dictionary)
- ▶ first simple program
- ▶ getting user input
- ▶ basic control structures

Control Structures Overview

- ▶ **if (condition):**
code block to execute if the condition is true
- ▶ **elif (other condition):**
code block to execute if the other condition is true
- ▶ **else:**
code block to execute if the condition(s) before are false
- ▶ **while (condition):**
code block to repeat until condition is true
- ▶ **for Variable in range (Start, Stop):**
code block to repeat for (Stop-Start) times

Indentation

In other programming languages blocks are enclosed for example by curly braces or by **begin** and **end**:

```
1 Block 1
2 if (val1 > val2)
3 {
4     Block 2
5 }
```

Python uses indentation for this:

```
1 Block 1
2 if (val1 > val2):
3     Block 2
```

Everything on the same indentation level belongs to the same block !

Indentation Examples

▶ example 1

```
1 sum = 0
2 for i in range(1,5):
3     sum += i
4     print sum
```

Indentation Examples

- ▶ example 1

```
1 sum = 0
2 for i in range(1, 5):
3     sum += i
4     print sum
```

- ▶ Throws an `IndentationError` as the indentation of the **print** statement doesn't match the indentation of any other block.

Indentation Examples

▶ example 1

```
1 sum = 0
2 for i in range(1,5):
3     sum += i
4     print sum
```

- ▶ Throws an IndentationError as the indentation of the **print** statement doesn't match the indentation of any other block.

▶ example 2

```
1 sum = 0
2 for i in range(1,5):
3     sum += i
4     print sum
```

Indentation Examples

▶ example 1

```
1 sum = 0
2 for i in range(1, 5):
3     sum += i
4     print sum
```

- ▶ Throws an `IndentationError` as the indentation of the **print** statement doesn't match the indentation of any other block.

▶ example 2

```
1 sum = 0
2 for i in range(1, 5):
3     sum += i
4     print sum
```

- ▶ The **print** statement belongs now to the block in the for loop and gets thus executed 4 times.

Indentation Examples (cont.)

▶ example 3

```
1 sum = 0
2 for i in range(1, 5):
3     sum += i
4 print sum
```

Indentation Examples (cont.)

▶ example 3

```
1 sum = 0
2 for i in range(1, 5):
3     sum += i
4 print sum
```

- ▶ The sum is only printed once, after the for loop has been executed. This is probably the intended version.

Insertion Sort - Short repetition

- ▶ You have two parts in your array: unsorted and sorted
- ▶ In the beginning the sorted part will probably include the first element, the rest is unsorted
- ▶ Every step you take the first element of the unsorted part and move it into the correct position of the sorted part
- ▶ What is the best case, what is the worst case ?



↩
insert v at the
correct position



Insertion Sort in python

```
1 myList = [4,8,3,5,1,2,7,11,9,2]
2 print myList
3 for i in range (1,len(myList)):
4     # take the first value of the unsorted part
5     value = myList[i];
6     j = i - 1;
7     # move value to the left
8     while ((j >= 0) and (myList[j] > value)):
9         myList[j + 1] = myList[j]
10        j = j - 1
11        #myList[j] < value, so insert after it
12        myList[j + 1] = value
13 print myList
```

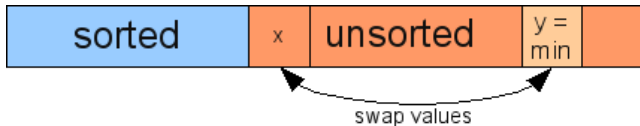
random lists and time measurements

- ▶ It would be interesting to see how fast the algorithm really is
- ▶ The list given in the previous slide was too small for any real time measurements.
- ▶ The next thing to do is to generate a list of a given size (via command line) with random numbers.
- ▶ The library **random** and **time** have what we want
- ▶ The function **random.seed()** initializes the random number generators, the function **random.random()** generates a random number between 0.0 and 1.0.
- ▶ For timing the function **time.clock()** returns the current processor time in seconds (as float)

```
1 import time,random,sys
2 random.seed()
3 if (len(sys.argv) > 1):
4     length = int(sys.argv[1])
5 else:
6     length = 1000
7 myList = []
8 for i in range (0,length):
9     myList.append(random.random())
10 start = time.clock ()
11 ....
12 # sorting myList
13 ....
14 end = time.clock()
15 print len(myList),"_", end - start
```

Selection Sort - Short repetition

- ▶ find minimum in the list
- ▶ swap with the value in the first position
- ▶ repeat the steps for the remainder of the list



Selection Sort in python

- ▶ This task is now up to you
- ▶ try to sort [8, 5, 1, 3, 9, 2, 5, 4, 12, 6] with your program
- ▶ If your program sorts it correct, use random lists and measure the running time (like in slide 16).
- ▶ generate lists that represent the best and worst cases and measure the running times
- ▶ I will present my solution next time
- ▶ you are free to send me your solution as well as your problems (e-mail adress: jenz@hlrs.de), but you aren't obligated to do so.

Functions

Sometimes you will see, that certain code pieces are needed more than once.

- ▶ Copy and Paste !
- ▶ Put the code piece in a function and call this function whenever it is needed

The first solution is usually not recommended (the code will get unmaintainable).

You have already used functions, that were already included with the python package.

Usual Form:

```
def functionName (parameters):
```

```
    Your code
```

```
    return someValue
```

Functions (cont.)

- ▶ A function can get parameters (you define how much).
- ▶ It always returns a value, that you specify with the **return** statement, after which the function terminates.
- ▶ If you don't have a **return** statement the special value **None** is returned.
- ▶ As caller you can ignore the return value.
- ▶ If you want to return multiple values, you can return a list, a tuple or a dictionary.
- ▶ If your functions is specified with **n** parameters, you have to call it with **n** parameters.
- ▶ The parameter references are passed "call by value".

Examples

```
1 def factorial (n):
2     loopMax = n
3     for i in range(1, loopMax):
4         n *= i
5     return n
6
7 for i in range (1,6):
8     print factorial (i), '->', i
```

This prints the factorials of the numbers 1 to 5.

The calculation of the factorial is done in the function *factorial*, which wants one parameter.

Examples (cont.)

```
1 def insertionSort (myList):
2     for i in range (1,len(myList)):
3         value = myList[i];
4         j = i - 1;
5         while ((j >= 0) and (myList[j] > value)):
6             myList[j + 1] = myList[j]
7             j = j - 1
8         myList[j + 1] = value
9 simpleList = [4,8,3,5,1,2,7,11,9,2]
10 print simpleList
11 insertionSort (simpleList)
12 print simpleList
```

Here the insertion sort from some slides before is put into a function, which expects a list as parameter.

What output do you expect ?

Recursion

"In order to understand recursion, one must first understand recursion."

- ▶ You can call functions inside of functions (not that surprising)
- ▶ A function can also call itself !
- ▶ In mathematics that's nothing new, e.g.

$$F_n = F_{n-1} + F_{n-2}$$

- ▶ The code could be:

```
1 def fibo (n):  
2     return fibo(n-1) + fibo(n-2)  
3 print fibo (5)
```

Recursion (cont.)

Fortunately python stops after some time with "RuntimeError: maximum recursion depth exceeded".

- ▶ Of course, we forgot something: The base cases (as termination condition)

$$F_0 = F_1 = 1$$

- ▶ Without it, the recursion would theoretically never stop.
- ▶ The modified code:

```
1 def fibo (n):
2     if (n < 2):
3         return 1
4     else:
5         return fibo(n-1) + fibo(n-2)
6 print fibo (5)
```

Recursion (cont.)

As seen a recursive function consists of two parts (from Wikipedia):

- ▶ Are we done yet? If so, return the results. Without such a termination condition a recursion would go on forever.
- ▶ If not, simplify the problem, solve the simpler problem(s), and assemble the results into a solution for the original problem. Then return that solution.

Recursion vs. Iteration

We have calculated the Fibonacci numbers in two ways now: iterative and recursive.

- ▶ Generally an iterative implementation will be faster than a recursive.
- ▶ That's because each function call has some overhead (e.g. setting up stacks)
- ▶ But sometimes there is no iterative implementation:
 - ▶ Traversal of a tree
 - ▶ Divide and Conquer algorithms (e.g. Quicksort)
 - ▶ functions like the Ackermann function

$$A(m, n) = \begin{cases} n + 1 & m = 0 \\ A(m - 1, 1) & m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & m > 0 \text{ and } n > 0 \end{cases}$$

Quicksort - repetition

Quicksort is a Divide and Conquer algorithm.

- ▶ determine an element in the list (pivot)
- ▶ reorder the list, that all elements smaller than the pivot will be left of the pivot, and all elements greater on the right.
- ▶ sort the elements to the left of the pivot element and those to the right (with Quicksort again)
- ▶ a list with only 0 or 1 element is, of course, sorted.



sort these with Quicksort

Homework:

- ▶ Program the Ackermann function (when you test it, don't choose the first parameter too high !)
- ▶ Implement the Quicksort algorithm.
- ▶ Problems ? Mail me at **jenz@hls.de** or visit me (Allmandring 30, room 0.032).