



# Introduction to Computer Science and Programming

## Lecture 3

Sebastian Wandelt

Beihang University

# Recap – What do you remember?



# Recap – What do you remember?



When I say PYTHON, do you think of ....

```
name=input("Your name:")
age=input("Your age:")
if age >= 18:
    print("Welcome!")
else:
    print("😞")
```

OR



?

# What else do you remember?

# What is wrong with this code?

```
a=int(input("a="))
b=int(input("b="))
c=int(input("c="))
if a!=0 and b**2-4*a*c>=0:
    x=(-b+(b**2-4*a*c)**0.5)/2*a
elif a=0:
    x=b
else:
    print("no real roots")
print(x)
```

# Outline

- Lists
- Loops
- Functions


# A new datatype: Lists

# More datatypes? Lists!

- We know three datatypes for variables
  - Strings, Floats, Integers
- Lists are an ordered collection of elements
- A variable with 0 or more things inside of it
- Examples
  - [1,2,3,4,5]
  - ["a", "b", "c", "d", "e"]
  - [1.0,1.1,9.4,2.6]
  - More examples:
    - [1,2, "a",1.7,9.5]
  - Even more examples:
    - [1,2, "a",[5,4, "z"]]



# More datatypes? Lists!

- We know three datatypes for variables
  - Strings, Floats, and Integers
- Lists are an ordered collection of elements
- A variable with 0 or more things inside of it
- Examples
  - [1,2,3,4,5]
  - ["a","b","c","d","e"]
  - [1.0,1.1,9.4,2.6]
  - More examples:
    - [1,2,"a",1.7,9.5]   
Lists can contain any (mixed) datatypes
  - Even more examples:
    - [1,2,"a",[5,4,"z"]]   
Lists can even contain lists!

# Creating a list

- Brackets [ ] are your friend
- Examples:

```
emptyList = []  
print(emptyList)  
groceries = ["milk", "eggs", "yogurt"]  
print(groceries)
```

# Elements and Indexing

- Each slot / thing in a list is called an “element”
- How many elements are in the list

$L = [3, 5, 7, 8]$ ?

- Each element has an “index” or location
- In  $L$  above,
  - 3 is found at the 0<sup>th</sup> index
  - 5 is at the 1<sup>th</sup> index
  - 7?
  - 8?

# Accessing elements

- Access individual elements using brackets
- Example:

```
L=[202,10, 54, 23]
```

```
print( L[1] + L[3] )
```

```
L[0] = 1    #changing an element
```

```
print(L)
```

What is the output?



# List length

- Use `len()` function to find out the size of a list

- Examples:

```
L = []
```

```
print(len(L))
```

```
L2 = [8, 6, 7, 5, 3, 0, 9]
```

```
print(len(L2))
```

# List slicing

- Access a slice (sub-list) using ranges Includes first element in range, excludes last in range
- Example:

```
L=[202,10, 54, 23]
```

```
print(L[0:1])
```

```
print(L[2:3])
```

```
print(L[0:0])
```

What is the output?



# Adding to a list

- Example

```
L = [5, 6, 87, 9, 2, 4]
```

```
print(L)
```

```
L.append(99999)
```

```
print(L)
```

# Deleting from a list

- Example

```
L = [5, 6, 87, 9, 2, 4]
```

```
print(L)
```

```
del L[2]
```

```
print(L)
```



# Lists: Summary

- Lists are a compound datatypes, built on top of simpler datatypes
- You will deal with lists very frequently (as a computer engineer)
- Basic operations are:
  - Setting elements
  - Retrieving elements
  - Iterating elements ...
- An overview an many operations:
  - <https://docs.python.org/3/tutorial/datastructures.html>
- Why is this important here?
  - Because lists can have arbitrary lengths!

# Repetitions/Loops

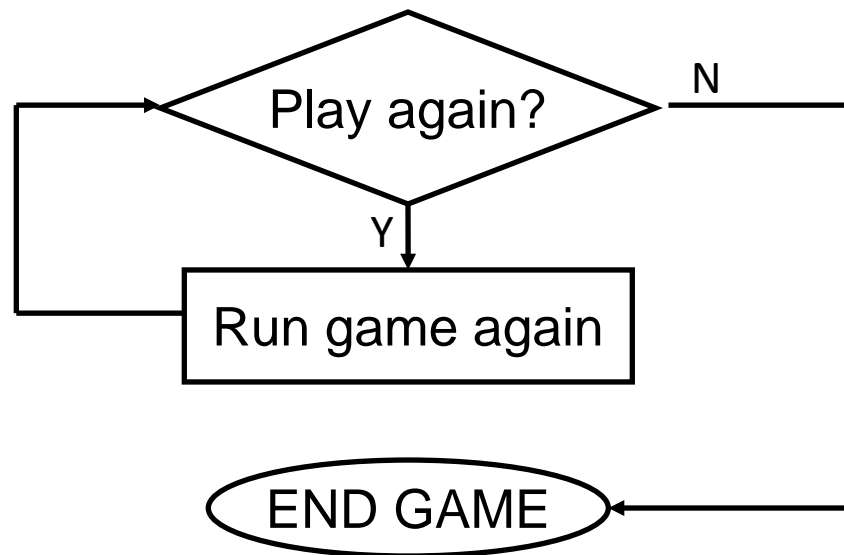
# Looping/Repetition

- How to get the program or portions of the program to automatically re-run, without duplicating the instructions!



Re-running the entire program

## Flowchart



## Pseudo code

While the player wants to play

Run the game again

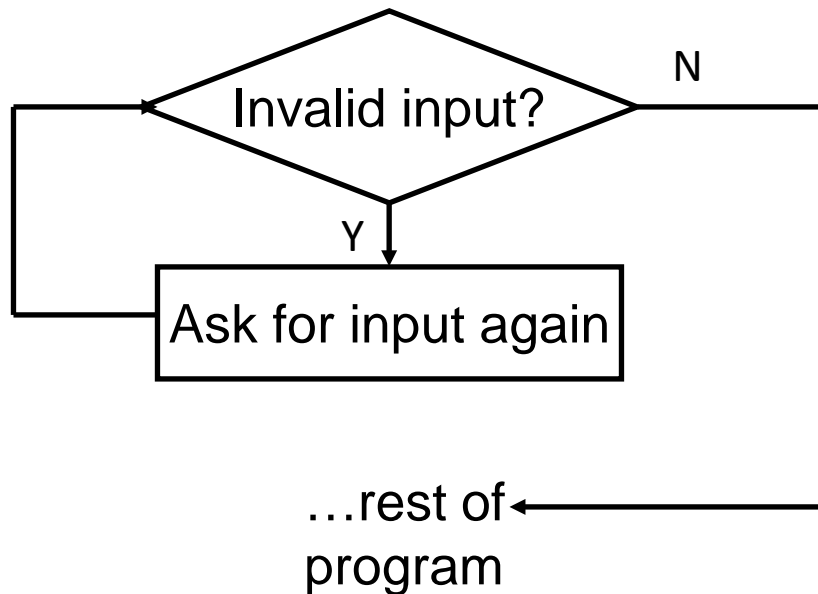
# Another example

- Example 2:

```
Enter your age (must be non-negative): -1
Enter your age (must be non-negative): 27
Enter your gender (m/f): 
```

Re-running specific parts of the program

## Flowchart



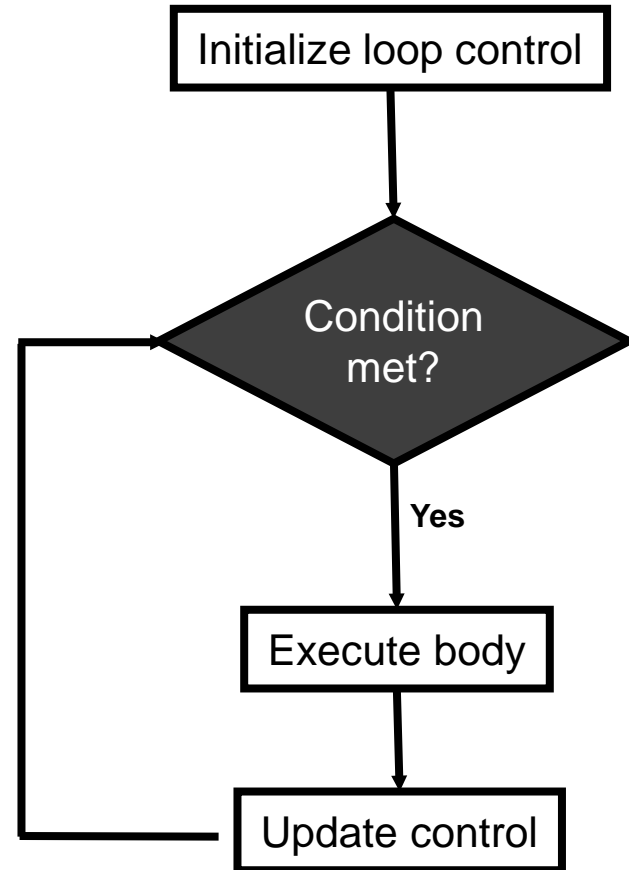
## Pseudo code

While input is invalid  
    Prompt user for input

# Basic Structure Of Loops

Whether or not a part of a program repeats is determined by a loop control (typically the control is just a variable).

1. Initialize the control to the starting value
2. Testing the control against a stopping condition (Boolean expression)
3. Executing the body of the loop (the part to be repeated)
4. Update the value of the control



# Two kinds of loops in Python

1. `while`
2. `for`

Both are pre-test loops!

## **Characteristics:**

1. The stopping condition is checked *before* the body executes.
2. These types of loops execute zero or more times.

# The while Loop

- This type of loop can be used if it's *not known* in advance how many times that the loop will repeat (most powerful type of loop, any other type of loop can be simulated with a while loop).
  - It can repeat so long as some arbitrary condition holds true.
- **Format:**

```
while (Boolean expression):  
    body
```

# The while Loop (2)

- **Example:**

```
i = 1
while (i <= 3):
    print("i =", i)
    i = i + 1
print("Done!")
```

1) Initialize control

2) Check condition

3) Execute body

4) Update control



# Countdown Loop

- **Example:**


```
i = 3
while (i >= 1):
    print("i =", i)
    i = i - 1
print("Done!")
```

# Common Mistakes: While Loops

- Forgetting to include the basic parts of a loop.

- Updating the control

```
i = 1
while(i <= 4):
    print("i =", i)
    # i = i + 1
```



```
i = 1
i = 1
i = 1
i = 1
i = 1
i = 1
i = 1
i = 1
i = 1
i = 1
```

# The for Loop

- In Python a for-loop is used to step through a sequence e.g., count through a series of numbers or step through the lines in a file.

- **Syntax:**

```
for <name of loop control> in <something that can be iterated>:  
    body
```

- **Example:**

```
total = 0  
for i in [1,2,3]:  
    total = total + i  
    print("i=", i, "\ttotal=", total)  
print("Done!")
```

1) Initialize control

2) Check condition

4) Update control

3) Execute body

# Counting Down With A For Loop

- **Example:**

```
i = 0
total = 0
for i in range (3, 0, -1):
    total = total + i
    print("i = ", i, ", total = ", total)
print("Done!")
```

# Loop Increments Need **Not Be Limited To One**

- **while:**

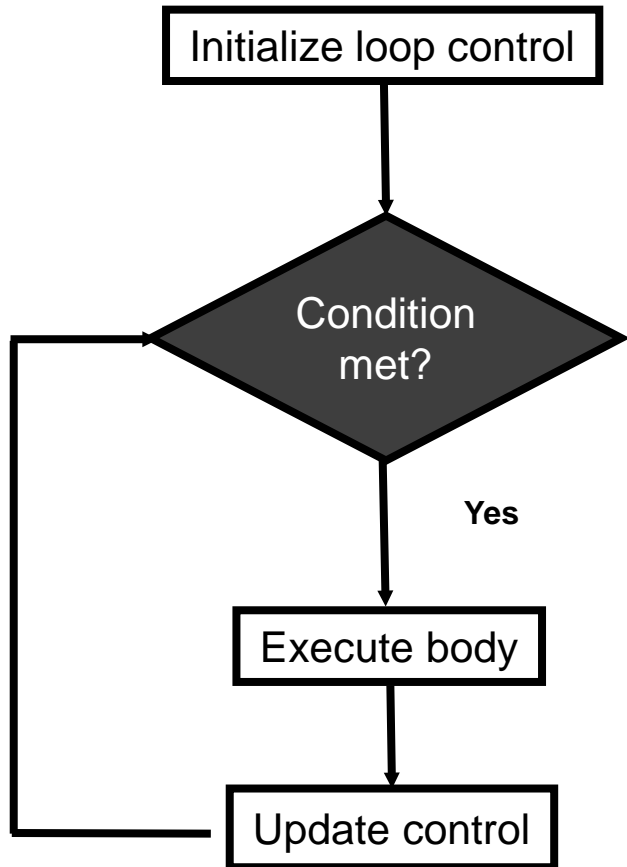
```
i = 0
while (i <= 100):
    print("i =", i)
    i = i + 5
print("Done!")
```

- **for:**

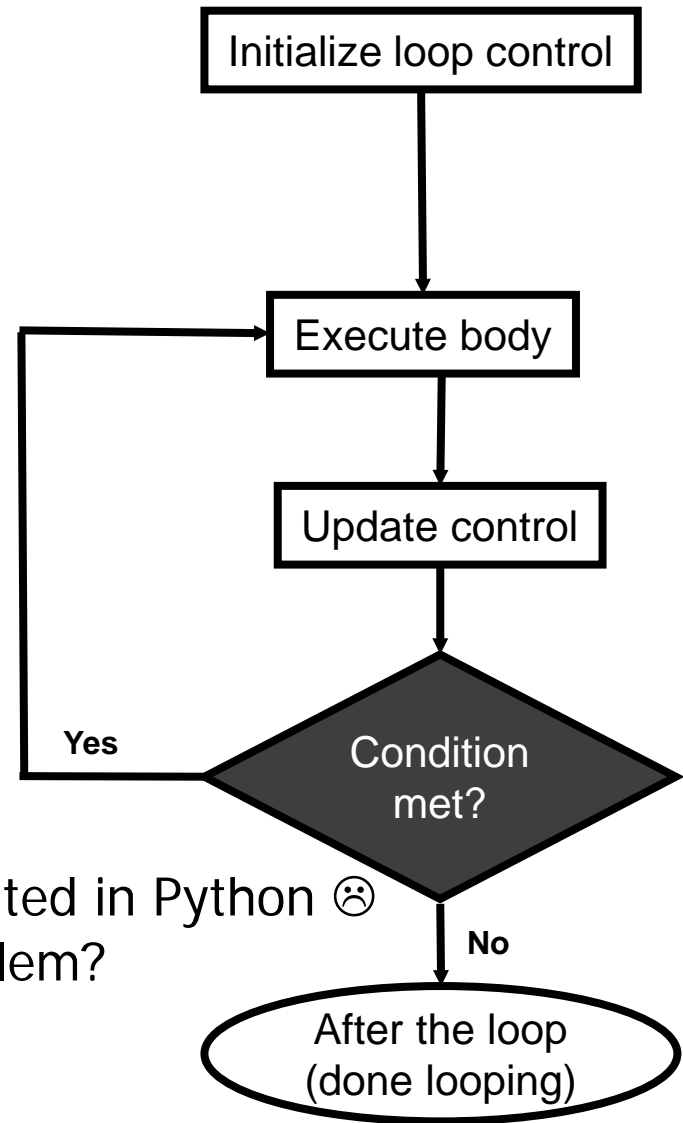
```
for i in range (0, 105, 5):
    print("i =", i)
print("Done!")
```

```
i = 0
i = 5
i = 10
i = 15
i = 20
i = 25
i = 30
i = 35
i = 40
i = 45
i = 50
i = 55
i = 60
i = 65
i = 70
i = 75
i = 80
i = 85
i = 90
i = 95
i = 100
Done!
```

# Pre-Test Loops vs Post-Test Loops



Not implemented in Python ☹️  
Is that a problem?



# Common Mistake #1

- Mixing up branches (IF and variations) vs. loops (while)
- Related (both employ a Boolean expression) but they are not identical
- Branches
  - General principle: If the Boolean evaluates to true then execute a statement or statements (**once**)
  - Example: display a popup message if the number of typographical errors exceeds a cutoff.
- Loops
  - General principle: As long as (or while) the Boolean evaluates to true then execute a statement or statements (**multiple times**)
  - Example: While there are documents in a folder that the program hasn't printed then continue to open another document and print it.

# Nesting

- Recall: Nested branches (one inside the other)

- Nested branches:

```
If (Boolean):  
    If (Boolean):  
        ...
```

- Branches and loops (for, while) can be nested within each other

```
# Scenario 1  
loop (Boolean):  
    if (Boolean):  
        ...  
  
# Scenario 2  
if (Boolean):  
    loop (Boolean):  
        ...
```

```
# Scenario 3  
loop (Boolean):  
    loop (Boolean):  
        ...
```



# Recognizing When Nesting Is Needed

- **Scenario 1:** As long some condition is met a question will be asked. As the question is asked if the answer is invalid then an error message will be displayed.
    - Example: While the user entered an invalid value for age (too high or too low) then if the age is too low an error message will be displayed.
    - Type of nesting: an IF-branch nested inside of a loop
- ```
loop (Boolean):  
    if (Boolean):  
        ...
```

# Recognizing When Nesting Is Needed

- **Scenario 2:** If a question answers true then check if a process should be repeated.
  - Example: If the user specified the country of residence as China then repeatedly prompt for the province of residence as long as the province is not valid.
  - Type of nesting: a loop nested inside of an IF-branch

If (Boolean):

    loop ():

        ...

# Recognizing When Nesting Is Needed

- **Scenario 3:** While one process is repeated, repeat another process.
  - More specifically: for each step in the first process repeat the second process from start to end
  - Example: While the user indicates that he/she wants to calculate another tax return prompt the user for income, while the income is invalid repeatedly prompt for income.
  - Type of nesting: a loop nested inside of an another loop

Loop():

    Loop():

        ...

# Practice Example #2: Nesting

1. Write a program that will count out all the numbers from one to six.
  2. For each of the numbers in this sequence the program will determine if the current count (1 – 6) is odd or even.
    - a) The program should display the value of the current count as well an indication whether it is odd or even.
- Which Step (#1 or #2) should be completed first?



# Step #1 Completed: Now What?

- For each number in the sequence determine if it is odd or even.
- This can be done with the modulo (remainder) operator: %
  - An even number modulo 2 equals zero (2, 4, 6 etc. even divide into 2 and yield a remainder or modulo of zero).
  - `if (counter % 2 == 0): # Even`
  - An odd number modulo 2 does not equal zero (1, 3, 5, etc.)
- Pseudo code visualization of the problem

Loop to count from 1 to 6

Determine if number is odd/even and display message

End Loop

- Determining whether a number is odd/even is a part of counting through the sequence from 1 – 6, checking odd/even is nested within the loop

# Step #2 Solution

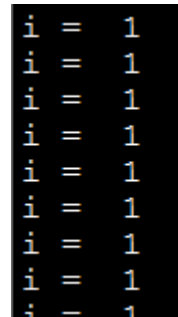
```
counter = 1
while (counter <= 6):
    if (counter % 2 == 0):
        print("Even:", counter)
    else:
        print("Odd:", counter)
    counter = counter + 1
```

# Infinite Loops

- Infinite loops never end (the stopping condition is never met).
- They can be caused by logical errors:
  - The loop control is never updated:

- **Example 1:**

```
i = 1
while (i <= 10):
    print("i = ", i)
    i = i + 1
```

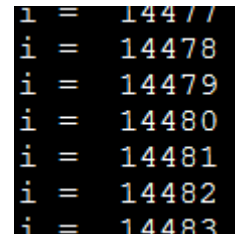


```
i = 1
i = 1
i = 1
i = 1
i = 1
i = 1
i = 1
i = 1
i = 1
i = 1
```

# Infinite Loops (2)

- Infinite loops never end (the stopping condition is never met).
- They can be caused by logical errors:
  - The updating of the loop control never brings it closer to the stopping condition:
- **Example 2:**

```
i = 10
while (i > 0):
    print("i = ", i)
    i = i + 1
print("Done!")
```



```
i = 14477
i = 14478
i = 14479
i = 14480
i = 14481
i = 14482
i = 14483
```



# Testing Loops

- Make sure that the loop executes the proper number of times.
- Test conditions:
  - 1) Loop does not run
  - 2) Loop runs exactly once
  - 3) Loop runs exactly **n** times

# After This Section You Should Know

- When and why are loops used in computer programs
- What is the difference between pre-test loops and post-test loops
- What types of loops exist in Python
- How to properly write the code for a loop in a program
- What are nested loops and how do you trace their execution

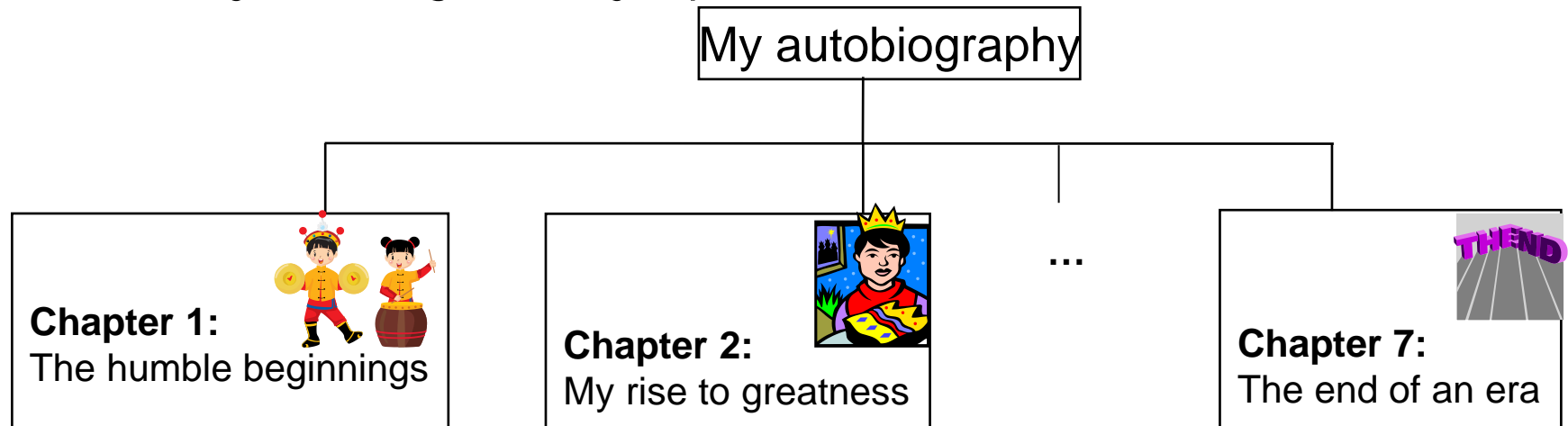
# Functions

# Solving Larger Problems

- Sometimes you will have to write a program for a large and/or complex problem.
- One technique employed in this type of situation is the top-down approach to design.
  - The main advantage is that it reduces the complexity of the problem because you only have to work on it a portion at a time.

# Top Down Design: Your life in a book


1. Start by outlining the major parts (structure)



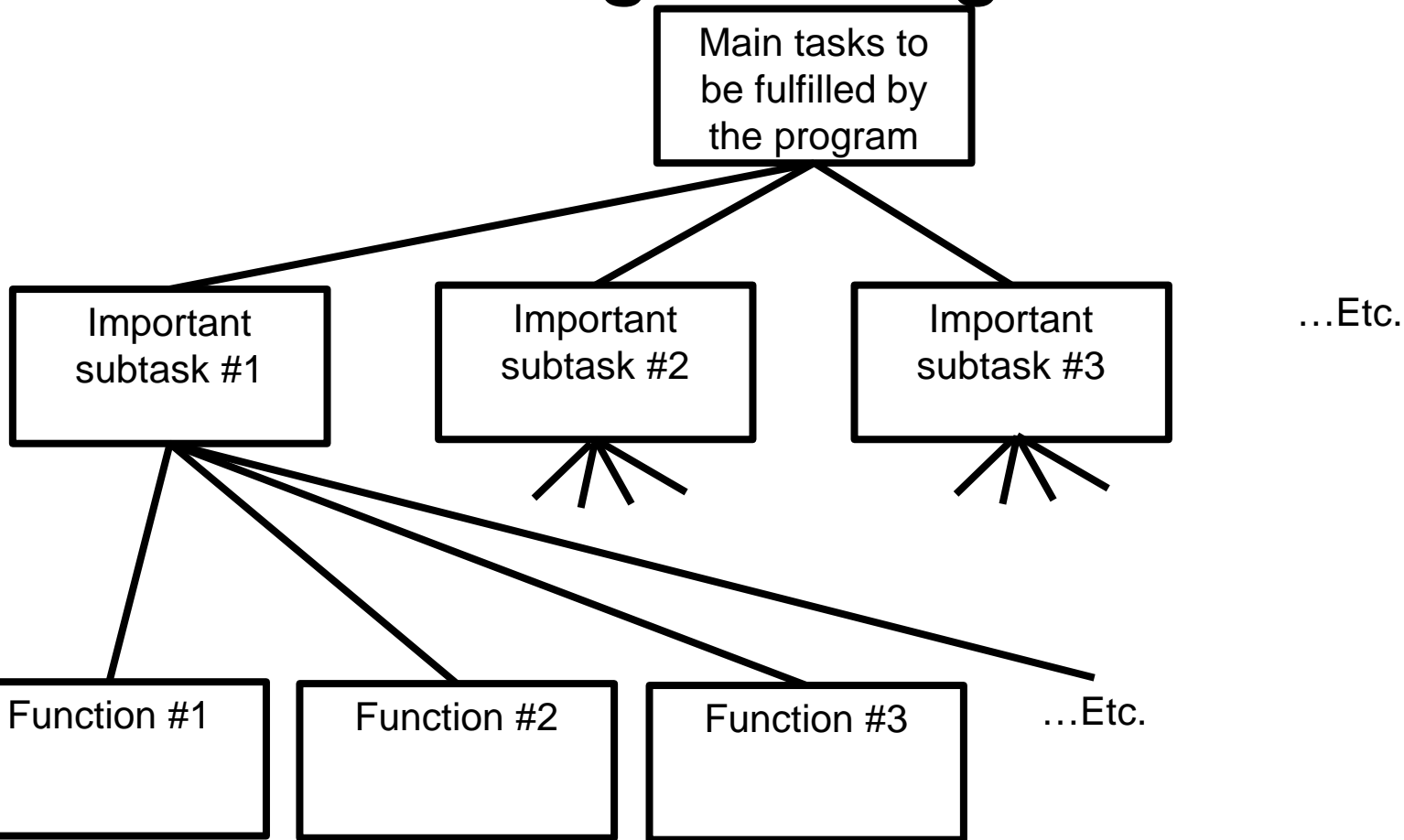
2. Then implement the solution for each part

**Chapter 1: The humble beginnings**

It all started ten and one score years ago with a log-shaped computer work station...



# Procedural Programming



When do you stop decomposing and start writing functions? No clear cut off but use your intuition as a guide e.g., a function should have one well defined task and not exceed one screen in length.

# Example Problem

- Design a program that will perform a simple interest calculation.
- The program should prompt the user for the appropriate values, perform the calculation and display the values onscreen.

How can we split this program/task into smaller subtasks?



# Example Problem

- Design a program that will perform a simple interest calculation.
- The program should **prompt** the user for the appropriate values, **perform the calculation** and **display** the values onscreen.
- Action/verb list:
  - Prompt
  - Calculate
  - Display



# Example code

```
print("Hello, I am Interest Calculator Version 1.")
initialAmount=float(input("How many Yuan do you have?"))
years=int(input("For how many years?"))
interest=0.1

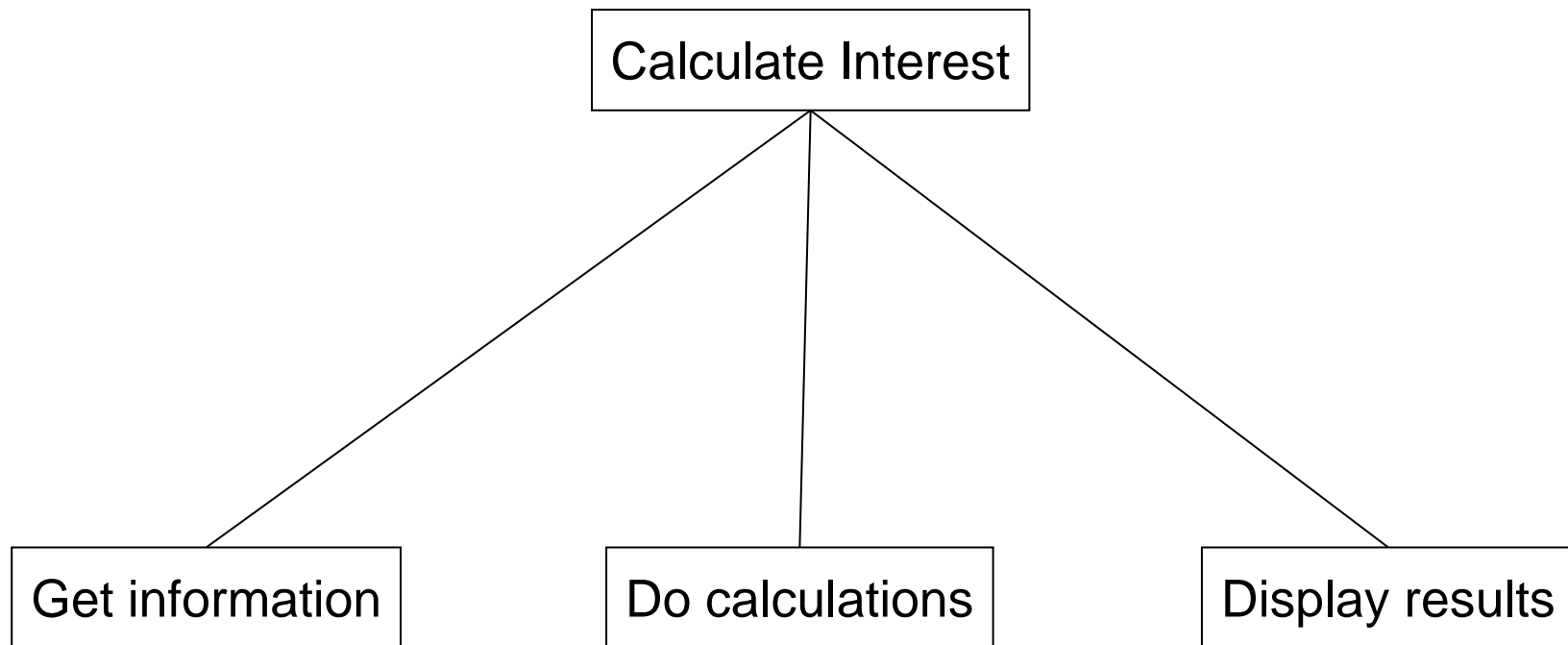
finalAmount=initialAmount
for y in range(years):
    finalAmount+=finalAmount*interest

print("Finally, you will have:",finalAmount,"Yuan")
print("That is a gain of:",finalAmount-initialAmount,"Yuan")
```

# Feedback from last year

- Now it becomes a bit tricky and confusing, although the concepts are quite natural **once** you understand them
- Please **listen carefully** and **ask** if you do not understand

# Top Down Approach: Breaking A Problem Down Into Parts (Functions)



# Example code with functions

```
def getInformation():  
    print("Hello, I am Interest Calculator Version 2.")  
    initialAmount=float(input("How many Yuan do you have?"))  
    years=int(input("For how many years?"))  
    interest=0.1  
    return (initialAmount,years,interest)  
  
def calculate(initialAmount,years,interest):  
    finalAmount=initialAmount  
    for y in range(years):  
        finalAmount+=finalAmount*interest  
    return finalAmount  
  
def displayResults(initialAmount,finalAmount):  
    print("Finally, you will have:",finalAmount,"Yuan")  
    print("That is a gain of:",finalAmount-initialAmount,"Yuan")  
  
initialAmount,years,interest=getInformation()  
finalAmount=calculate(initialAmount,years,interest)  
displayResults(initialAmount,finalAmount)
```

# Example code – step by step

```
def getInformation():  
    print("Hello, I am Interest Calculator Version 2.")  
    initialAmount=float(input("How many Yuan do you have?"))  
    years=int(input("For how many years?"))  
    interest=0.1  
    return (initialAmount,years,interest)
```

- The function name is: **getInformation**
- The function returns three variables: **initialAmount**, **years**, and **interest**
  - These variables can be used outside the function!

# Example code – step by step

```
def calculate(initialAmount, years, interest):  
    finalAmount = initialAmount  
    for y in range(years):  
        finalAmount += finalAmount * interest  
    return finalAmount
```

- The function name is: **calculate**
- The function receives three parameters: **initialAmount**, **years**, and **interest**
- The function returns one variable: **finalAmount**
  - This variable can be used outside the function!

# Example code – step by step

```
def displayResults(initialAmount,finalAmount):  
    print("Finally, you will have:",finalAmount,"Yuan")  
    print("That is a gain of:",finalAmount-initialAmount,"Yuan")
```

- The function name is: **displayResults**
- The function receives two parameters: **initialAmount** and **finalAmount**
- The function returns no results

# Example code – step by step

```
initialAmount,years,interest=getInformation()  
finalAmount=calculate(initialAmount,years,interest)  
displayResults(initialAmount,finalAmount)
```

- This is our main program now
  - Only three statements
  - Each statement calls a function and (possibly) assigns the result to variable names
- The structure is very clear



# Example code: Overview (again)

```
def getInformation():  
    print("Hello, I am Interest Calculator Version 2.")  
    initialAmount=float(input("How many Yuan do you have?"))  
    years=int(input("For how many years?"))  
    interest=0.1  
    return (initialAmount,years,interest)  
  
def calculate(initialAmount,years,interest):  
    finalAmount=initialAmount  
    for y in range(years):  
        finalAmount+=finalAmount*interest  
    return finalAmount  
  
def displayResults(initialAmount,finalAmount):  
    print("Finally, you will have:",finalAmount,"Yuan")  
    print("That is a gain of:",finalAmount-initialAmount,"Yuan")  
  
initialAmount,years,interest=getInformation()  
finalAmount=calculate(initialAmount,years,interest)  
displayResults(initialAmount,finalAmount)
```

# Function definition

Defining a function with parameters:

- **Format:**

```
def <function name>(<parameter 1>, <parameter 2>...  
    <parameter n-1>, <parameter n>):
```

- **Example:**

```
def calculate(initialAmount, years, interest):
```

Calling a function with parameters:

- **Format:**

```
<function name>(<parameter 1>, <parameter 2>...  
    <parameter n-1>, <parameter n>)
```

- **Example:**

```
calculate(10000,5,0.1)
```

# Using Return Values

- **Format (Multiple values returned):**

```
# Function definition
```

```
return(<value1>, <value 2>...)
```

```
# Function call
```

```
<variable 1>, <variable 2>... = <function name>()
```

- **Example (Multiple values returned):**

```
# Function definition
```

```
return (initialAmount,years,interest)
```

```
# Function call
```

```
initialAmount,years,interest=getInformation()
```

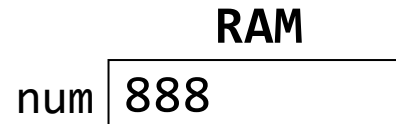
# New Terminology

- **Local variables:** are created within the body of a function (indented)
- **Global variables/constants:** created outside the body of a function.
- (The significance of global vs. local is coming up shortly).

# What You Know: Declaring Variables

- Variables are memory locations that are used for the temporary storage of information.

num = 888



- Each variable uses up a portion of memory, if the program is large then many variables may have to be declared (a lot of memory may have to be allocated to store the contents of variables).

# What You Will Learn: What Is The Significance Of Being 'Local'

- To minimize the amount of memory that is used to store the contents of variables only create variables when they are needed ("allocated").
- When the memory for a variable is no longer needed it can be 'freed up' and reused ("de-allocated").
- To design a program so that memory for variables is only allocated (reserved in memory) as needed and de-allocated when they are not (the memory is free up) variables should be declared as local to a function.

**Thank you very much!**