



Lecture 3

Lists, Loops, and Functions

Xiaoqian Sun / Sebastian Wandelt

Beihang University

Outline

- Recap
- Lists
- Loops
- Functions

A new datatype: Lists

More datatypes? Lists!

- We know three datatypes for variables
 - Strings, Floats, Integers
- Lists are an ordered set 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, Integers
- Lists are an ordered set 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

`myList = [3, 5, 7, 8]`?

- Each element has an “index” or location
- In the `myList` above,
 - 3 is found at the 0th index
 - 5 is at the 1th index
 - 7?
 - 8?

Accessing elements

- Access individual elements using brackets
- Example:

```
int_list=[202,10, 54, 23]
print( int_list[1] + int_list[3] )
int_list[0] = 1    #changing an element
print(int_list)
```


List length

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

```
myList = []  
print(len(myList))
```

```
myList2 = [8, 6, 7, 5, 3, 0, 9]  
print(len(myList2))
```

List slicing

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

- Example:

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

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

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

```
print(int _list[0:0])
```

- Omit values to get start or end of list

```
int_list[:2]
```

```
int_list[2:]
```

Adding to a list

- Example

```
myList = [5, 6, 87, 9, 2, 4]  
print(myList)  
myList.append(99999)  
print(myList)
```

Deleting from a list

- Example

```
myList = [5, 6, 87, 9, 2, 4]  
print(myList)  
del myList[2]  
print(myList)
```

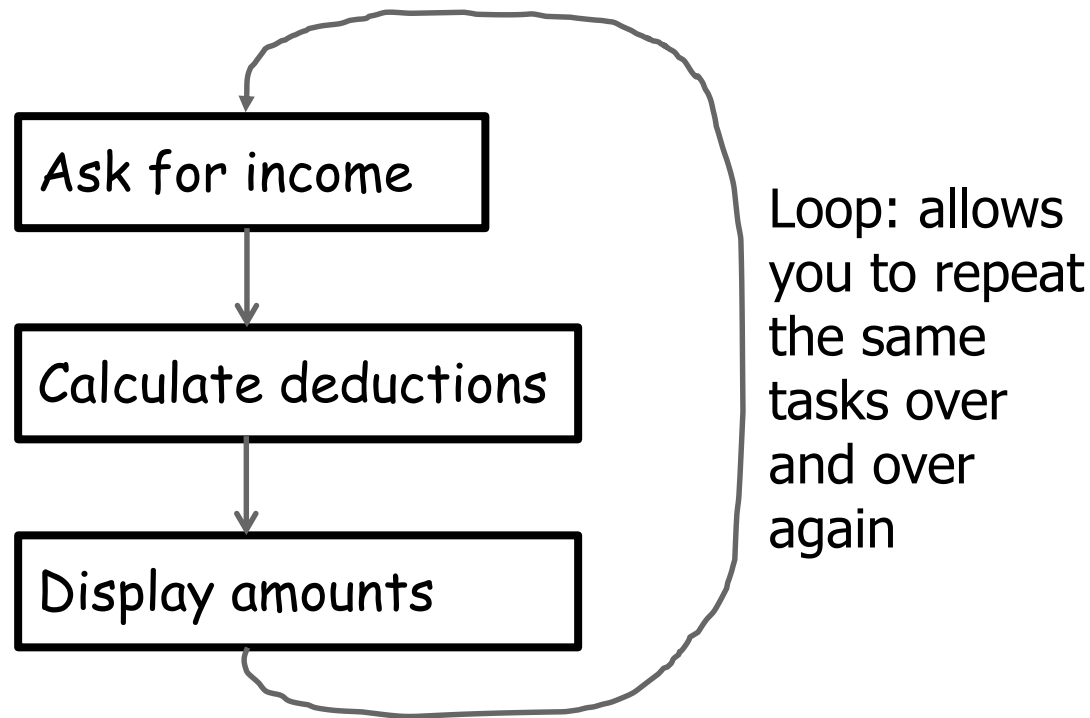
Lists: Summary

- 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 ...

Repetitions/Loops

Looping/Repetition

- How to get the program or portions of the program to automatically re-run
 - Without duplicating the instructions
 - Example: you need to calculate tax for multiple people



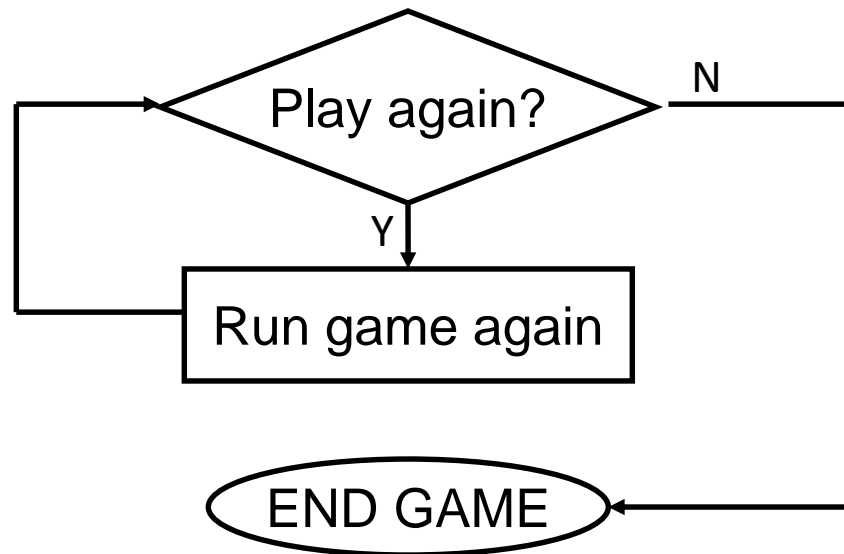
How To Determine If Loops Can Be Applied

- Something needs to occur multiple times (generally it will repeat itself as long as some condition has been met).
- Example 1:



Re-running the entire program

Flowchart



Pseudo code

While the player wants to play

Run the game again

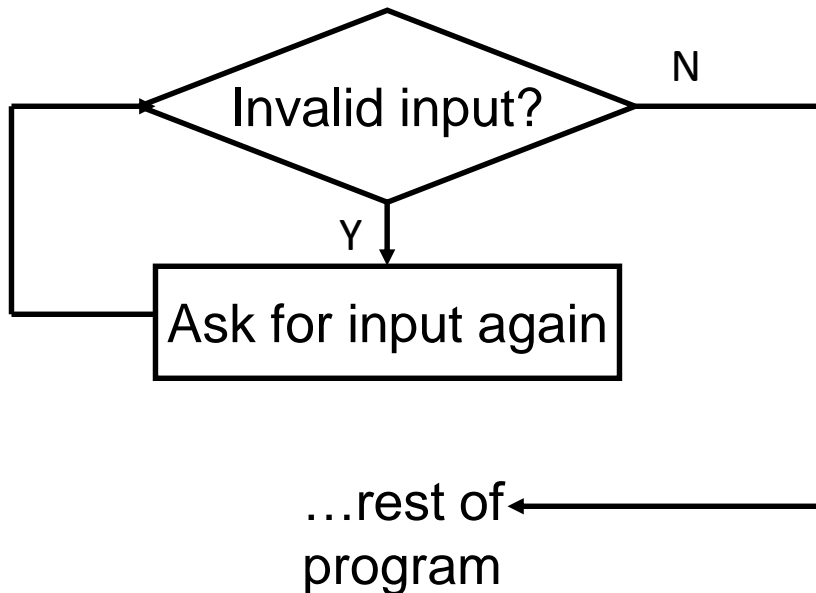
How To Determine If Loops Can Be Applied (2)

- 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

Types Of Loops

1. Pre-test loops

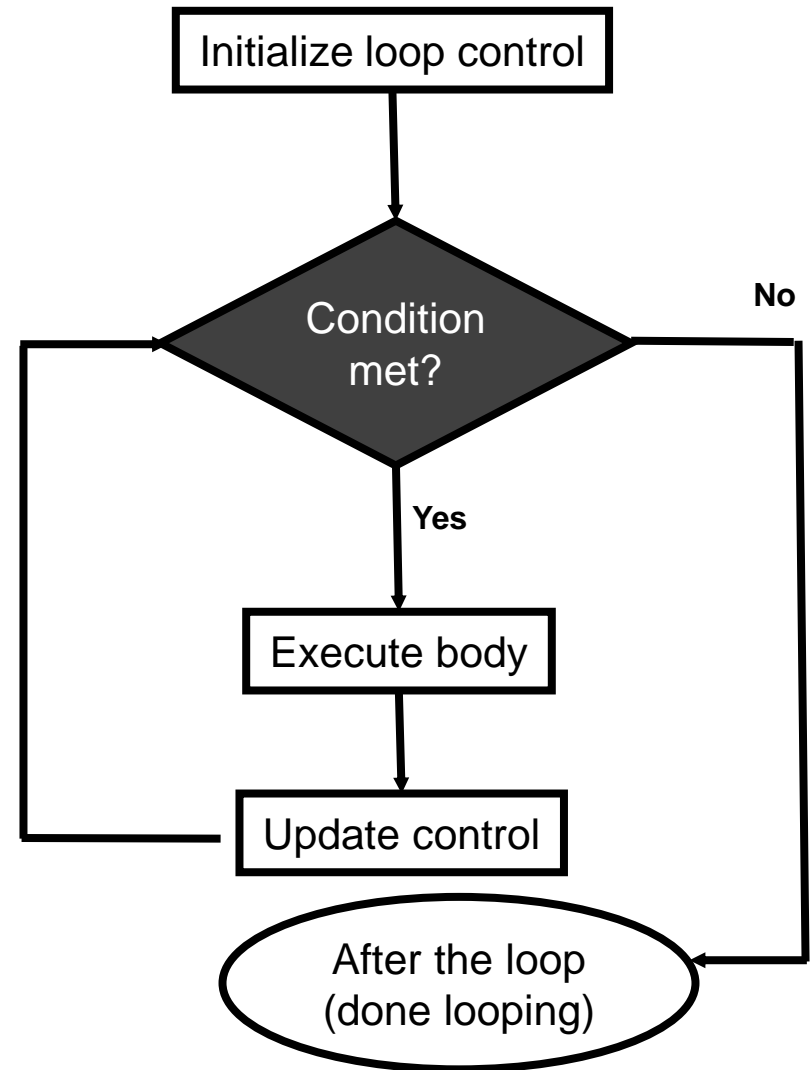
- Check the stopping condition *before* executing the body of the loop.
- The loop executes **zero or more** times.

2. Post-test loops

- Checking the stopping condition *after* executing the body of the loop.
- The loop executes **one or more** times.

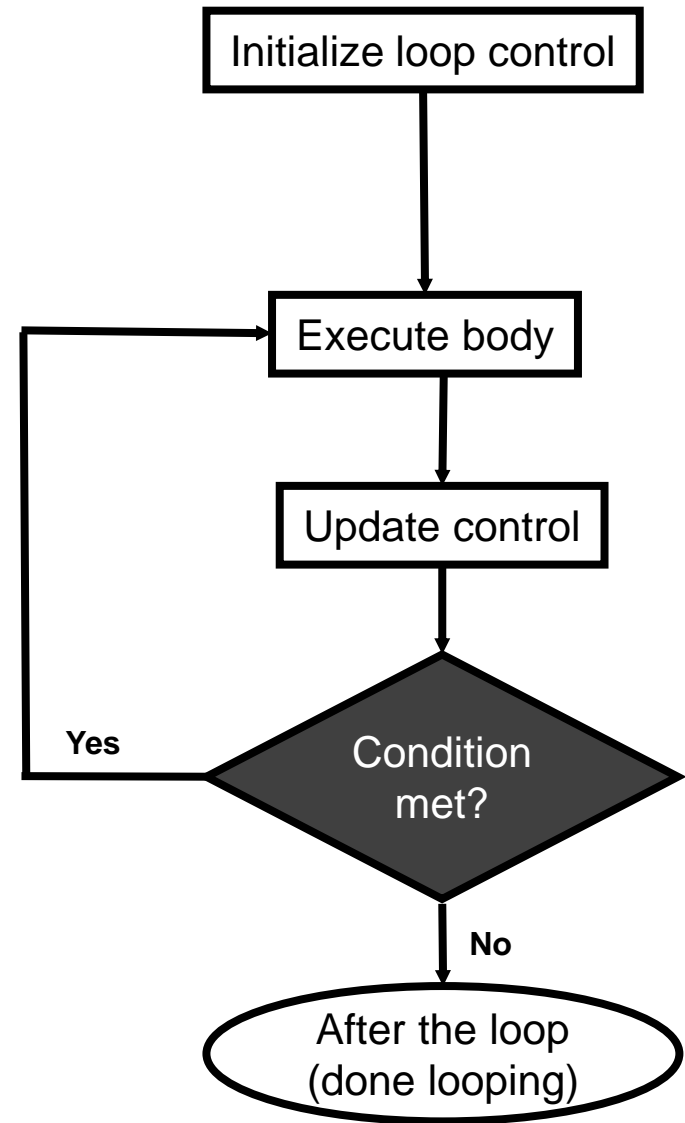
Pre-Test Loops

1. Initialize loop control
2. Check if the repeating condition has been met
 - a. If it's been met then go to Step 3
 - b. If it hasn't been met then the loop ends
3. Execute the body of the loop (the part to be repeated)
4. Update the loop control
5. Go to step 2



Post-Test Loops (Not Implemented In Python)

1. Initialize loop control (sometimes not needed because initialization occurs when the control is updated)
2. Execute the body of the loop (the part to be repeated)
3. Update the loop control
4. Check if the repetition condition has been met
 - a. If the condition has been met then go through the loop again (go to Step 2)
 - b. If the condition hasn't been met then the loop ends.



Pre-Test Loops In Python

1. While
2. For

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)

- **Program name:** while1.py

```
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

- **Program name:** while2.py


```
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
```

Practice Exercise #1

- The following program prompts for and displays the user's age.
- Modifications:
 - As long as the user enters a negative age the program will continue prompting for age.
 - After a valid age has been entered then stop the prompts and display the age.

```
age = int(input("Age: "))  
print(age)
```

```
Age: -1  
Age cannot be negative  
Age: -123  
Age cannot be negative  
Age: 27  
You are 27 years young :P
```



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 range (1, 4, 1):  
    total = total + i  
    print("i=", i, "total=", total)  
print("Done!")
```

1) Initialize control

2) Check condition

4) Update control

3) Execute body

Counting Down With A For Loop

- **Program name:** for2.py

```
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:** while_increment5.py

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

- **For:** for_increment5.py

```
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!
```

Recap: What Looping Constructs Are Available In Python/When To Use Them

Construct	When To Use
Pre-test loops	You want the stopping condition to be checked before the loop body is executed (typically used when you want a loop to execute zero or more times).
<ul style="list-style-type: none">• While	<ul style="list-style-type: none">• The most powerful looping construct: you can write a ‘while’ loop to mimic the behavior of any other type of loop. In general it should be used when you want a pre-test loop which can be used for most any arbitrary stopping condition e.g., execute the loop as long as the user doesn’t enter a negative number.
<ul style="list-style-type: none">• For	<ul style="list-style-type: none">• In Python it can be used to step through some sequence
Post-test: None in Python	You want to execute the body of the loop before checking the stopping condition (typically used to ensure that the body of the loop will execute at least once). The logic can be simulated with a while loop.

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():

 ...

Analyzing Another Nested Loop

- One loop executes inside of another loop(s).
- **Example structure:**
 - Outer loop (runs n times)
 - Inner loop (runs m times)
 - Body of inner loop (runs n x m times)

- Program name: nested.py

```
i = 1
while (i <= 2):
    j = 1
    while (j <= 3):
        print("i = ", i, " j = ", j)
        j = j + 1
    i = i + 1
print("Done!")
```

```
i = 1  j = 1
i = 1  j = 2
i = 1  j = 3
i = 2  j = 1
i = 2  j = 2
i = 2  j = 3
Done!
```

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 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

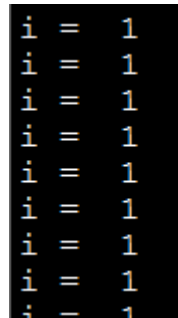
End Loop

- 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

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 – below).
 - The updating of the loop control never brings it closer to the stopping condition (Example 2 – next slide).
- **Example 1:** infinite1.py

```
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)

- **Example 2:** infinite2.py

```
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 Now Know

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

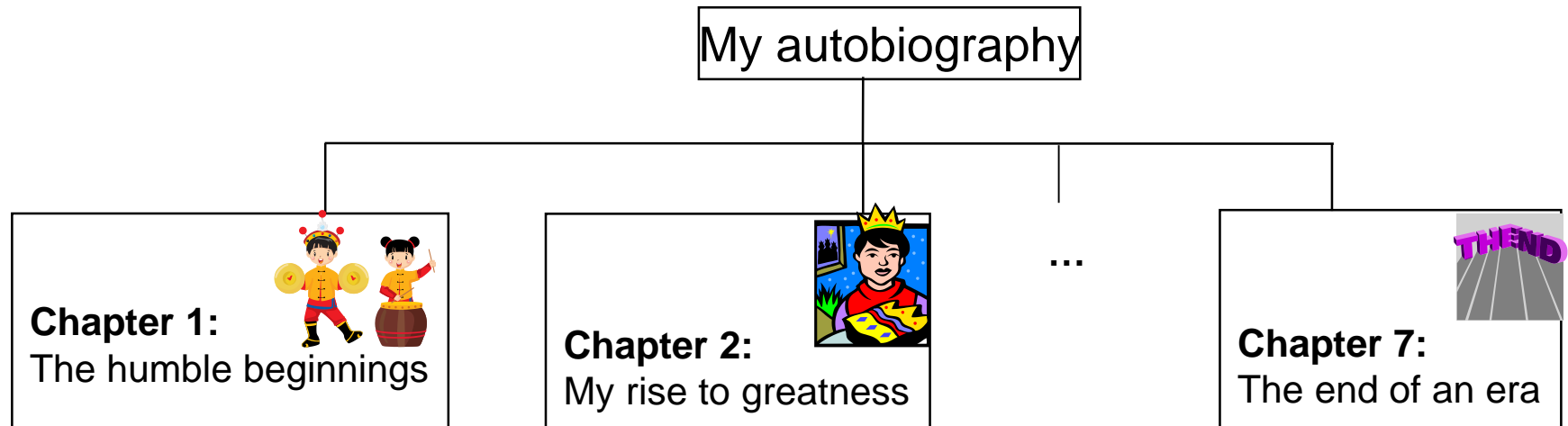
Functions

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

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 thousand years ago with a log-shaped computer work station...

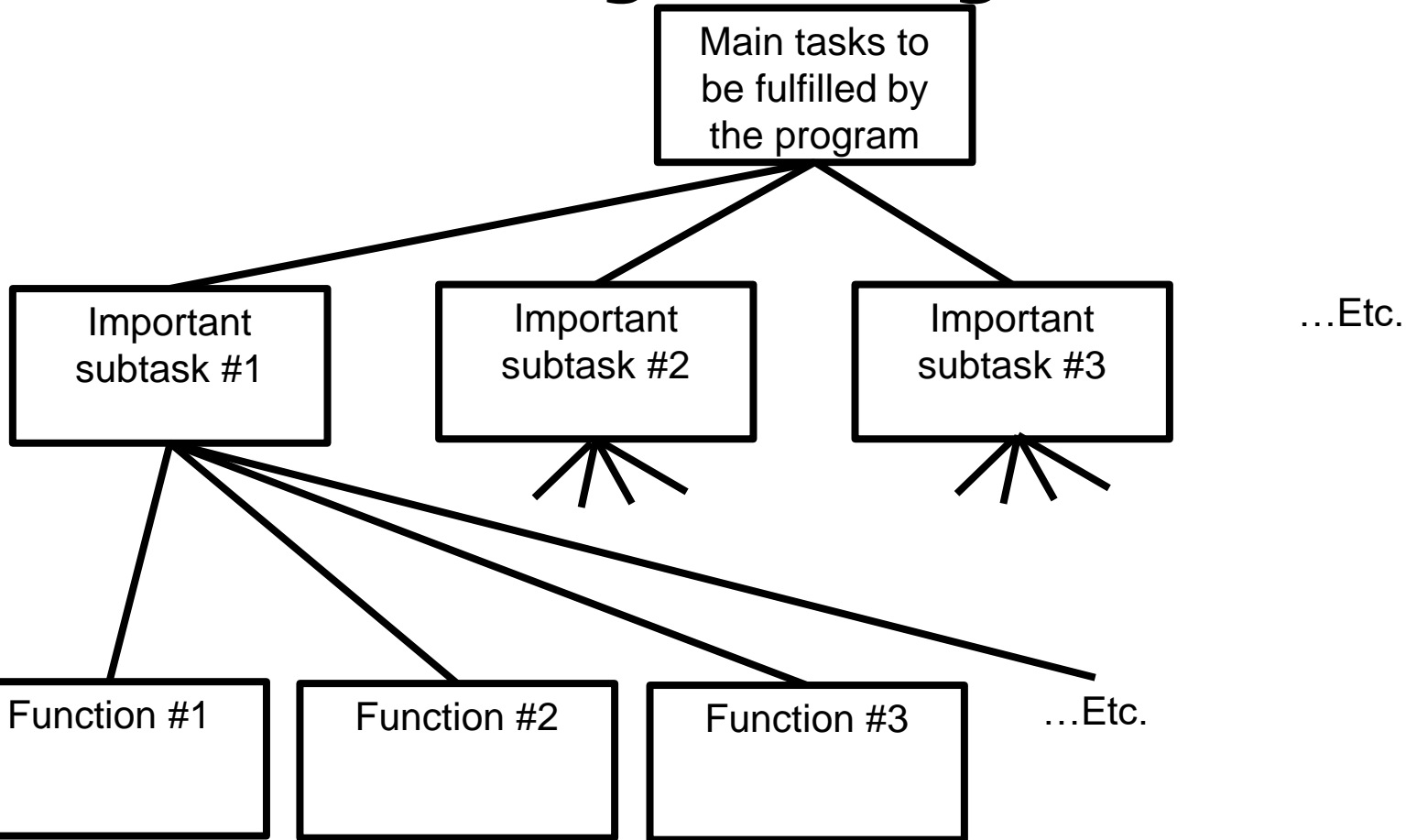


Image copyright unknown

Procedural Programming

- Applying the top down approach to programming.
- Rather than writing a program in one large collection of instructions the program is broken down into parts.
- Each of these parts are implemented in the form of procedures (also called “functions”, “procedures” or “methods” depending upon the programming language).

Procedural Programming



When do you stop decomposing and start writing functions? No clear cut off but use the “Good style” principles (later in these notes) as a guide e.g., a function should have one well defined task and not exceed a screen in length.

Decomposing A Problem Into Functions

- Break down the program by what it does (described with *actions/verbs* or *action phrases*).
- Eventually the different parts of the program will be implemented as functions.

Example Problem

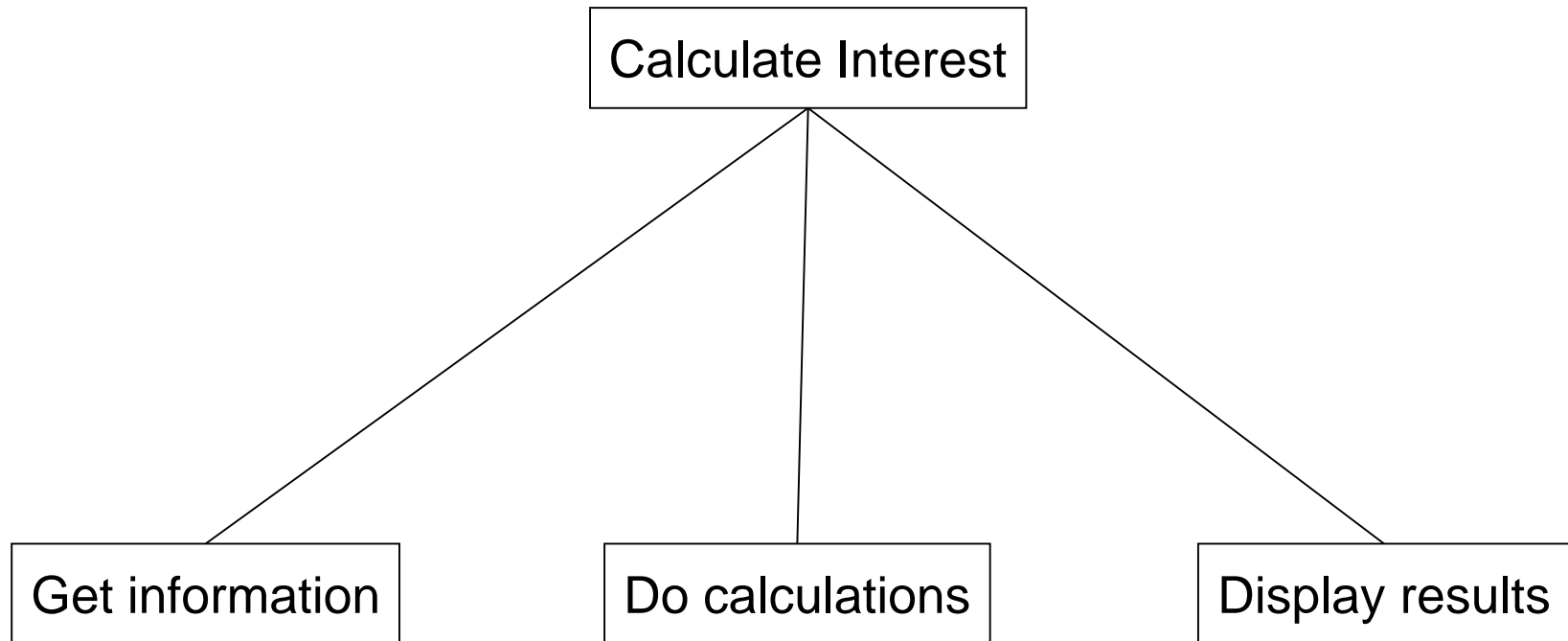
- Design a program that will perform a simple interest calculation.
- The program should prompt the user for the appropriate values, perform the calculation and display the values onscreen.



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

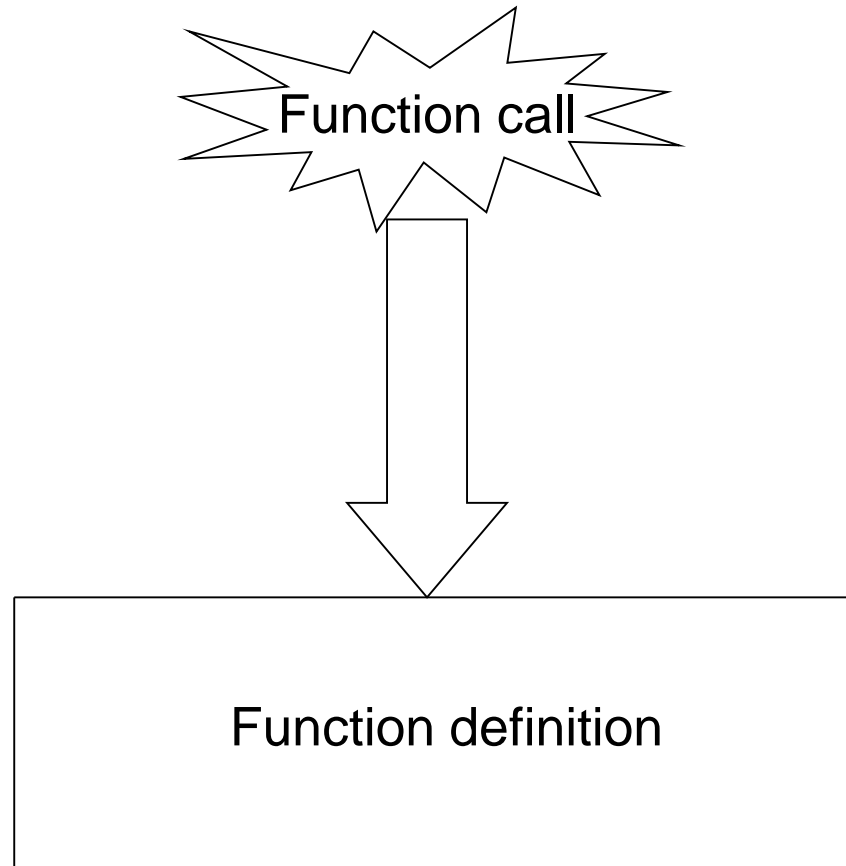
Top Down Approach: Breaking A Programming Problem Down Into Parts (Functions)



Things Needed In Order To Use Functions

- Function definition
 - Instructions that indicate what the function will do when it runs.
- Function call
 - Actually running (executing) the function.
 - You have already done this second part many times because up to this point you have been using functions that have already been defined by someone else e.g., `print()`, `input()`

Functions (Basic Case: No parameters/Inputs)



Defining and Calling a Function

- **Definition Format:**

```
def <function name>():  
    body1
```

- **Example:**

```
def displayInstructions():  
    print ("Displaying instructions on how to use the program")
```

- **Calling Format:**

```
<function name>()
```

- **Example:**

```
displayInstructions()
```

¹ Body = the instruction or group of instructions that execute when the function executes (when called).

The rule in Python for specifying the body is to use indentation.

Quick Recap: Starting Execution Point

- The program starts at the first executable instruction that is not indented.
- In the case of your programs thus far all statements have been unindented (save loops/branches) so it's just the first statement that is the starting execution point.

```
HUMAN_CAT_AGE_RATIO = 7
age = input("What is your age in years: ")
catAge = age * HUMAN_CAT_AGE_RATIO
...
```

- But note that the body of functions **MUST** be indented in Python.

Functions: An Example That Puts Together All The Parts Of The Easiest Case

- Name of the example program: `1firstExampleFunction.py`

```
def displayInstructions():  
    print("Displaying instructions")
```



Displaying instructions

Main body of code (starting execution point, not indented)

```
displayInstructions()  
print("End of program")
```

End of program

Functions: An Example That Puts Together All The Parts Of The Easiest Case

- Name of the example program: 1firstExampleFunction.py

```
def displayInstructions():  
    print("Displaying instructions")
```

**Function
definition**



Main body of code (starting execution point)

```
displayInstructions()  
print("End of program")
```

Function call



New Terminology

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

```
HUMAN_CAT_AGE_RATIO = 7

def getInformation():
    age = input("What is your age in years: ")
    catAge ← age * HUMAN_CAT_AGE_RATIO
```

Global
constant

Local
variables

Creating Your Variables

- Before: all statements (including the creation of a variables) occur outside of a function

```
HUMAN_CAT_AGE_RATIO = 7
age = input("What is your age in years: ")
catAge = age * HUMAN_CAT_AGE_RATIO
...
```

- Now that you have learned how to define functions, ALL your variables must be created with the body of a function.
- Constants can still be created outside of a function (more on this later).

```
HUMAN_CAT_AGE_RATIO = 7

def getInformation():
    age = input("What is your age in years: ")
    catAge = age * HUMAN_CAT_AGE_RATIO
```

**'Outside': OK for
constants only**

**Inside function
body: all variables
must be here**

What You Will Learn: How To Work With Locals

Function call (*local variables get allocated in memory*)

Function ends (*local variables get de-allocated in memory*)

The program code in the function executes
(the variables are used to store
information needed for the function)

Working With Local Variables: Putting It All Together

- Name of the example program: 3secondExampleFunction.py

```
def fun():
```

```
    num1 = 1
```

```
    num2 = 2
```

```
    print(num1, num2)
```

**Variables that
are local to
function 'fun'**



```
# start function
```

```
fun()
```

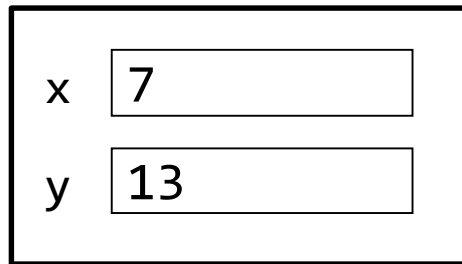
```
[csc decomposition 62 ]> python secondExampleFunction.py  
1 2
```

Local Variables

- Recall: local variables only exist for the duration of a function.
- After a function ends the local variables are no longer accessible.
- Benefit: reduces accidental changes to local variables.

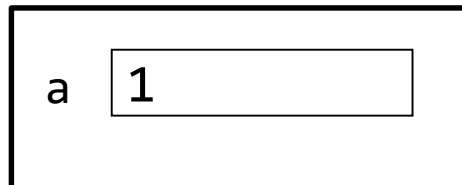
RAM

Memory: 'fun'



```
def fun():  
    x = 7  
    y = 13
```

Memory: 'start'



```
def start():  
    a = 1  
    fun()  
    # x,y  
    # inaccessible
```

```
start()
```

New Problem: Local Variables Only Exist Inside A Function

```
def display ():  
    print ("")  
    print ("Celsius value: ", celsius)  
    print ("Fahrenheit value :", fahrenheit)
```

What is 'celsius'???
What is 'fahrenheit'???

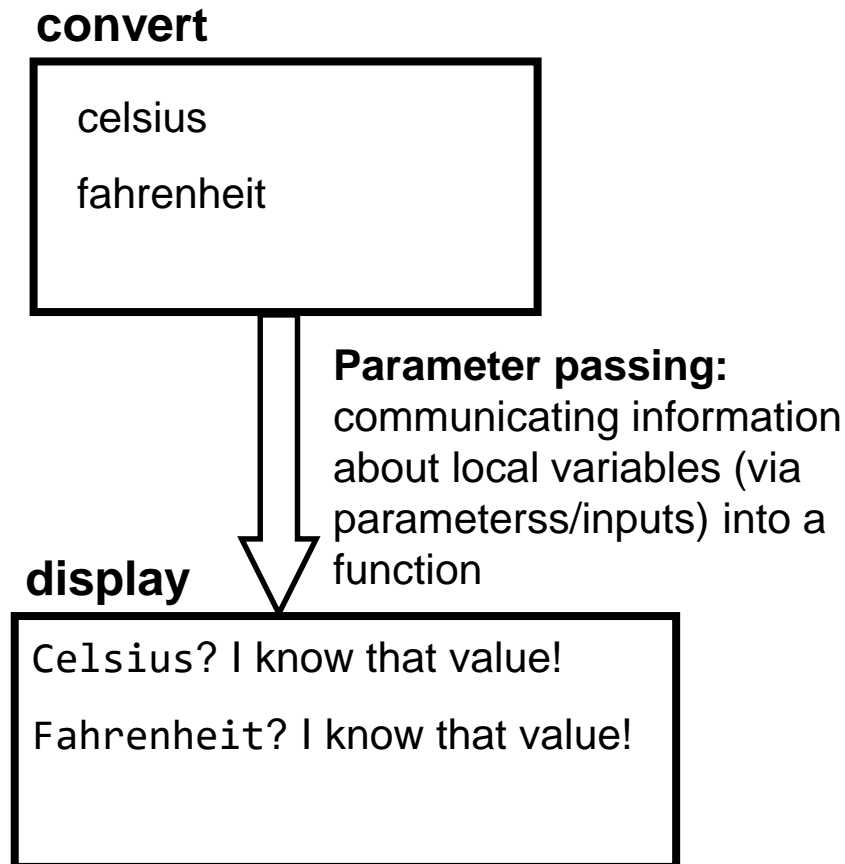
```
def convert ():  
    celsius ← float(input ("Type in the celsius temperature: "))  
    fahrenheit = celsius * 9 / 5 + 32  
    display ()
```

Variables celsius
and fahrenheit
are local to
function
'convert()'

New problem: How to access local variables outside of a function?

One Solution: Parameter Passing

- Passes a copy of the contents of a variable as the function is called:



Parameter Passing: Past Usage

- You did it this way so the function 'knew' what to display:

```
age = 27
```

```
# Pass copy of 27 to
```

```
# print() function
```

```
print(age)
```

- You wouldn't do it this way:

```
age = 27
```

```
# Nothing passed to print
```

```
# Function print() has
```

```
# no access to contents
```

```
# of 'age'
```

```
print()
```

```
# Q: Why doesn't it
```

```
# print my age?!
```

```
# A: Because you didn't
```

```
# tell it to!
```

Parameter Passing

- **Definiton Format:**

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

- **Example:**

```
def display(celsius, fahrenheit):
```

- **Call Format:**

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

- **Example:**

```
display(celsius, fahrenheit)
```

Memory And Parameter Passing

- Parameters passed as parameters/inputs into functions become variables in the local memory of that function.

```
def fun(num1):  
    print(num1)  
    num2 = 20  
    print(num2)
```

Parameter num1: local to fun

num2: local to fun

```
def start():  
    num1 = 1  
    fun(num1)
```

```
start()
```

num1: local to start

The Type And Number Of Parameters Must Match!

- **Correct 😊:**

```
def fun1(num1, num2):  
    print(num1, num2)
```

```
def fun2(num1, str1):  
    print(num1, str1)
```

```
# start
```

```
def start():  
    num1 = 1  
    num2 = 2  
    str1 = "hello"  
    fun1(num1, num2)  
    fun2(num1, str1)
```

```
start()
```

Two parameters (a number and a string) are passed into the call for 'fun2()' which matches the type for the two parameters listed in the definition for function 'fun2()'

Two numeric parameters are passed into the call for 'fun1()' which matches the two parameters listed in the definition for function 'fun1()'

A Common Mistake: The Parameters Don't Match

- **Incorrect ☹:**

```
def fun1(num1):  
    print(num1, num2)
```

```
def fun2(num1, num2):  
    num1 = num2 + 1  
    print(num1, num2)
```

```
# start
```

```
def start():  
    num1 = 1  
    num2 = 2  
    str1 = "hello"  
    fun1(num1, num2)  
    fun2(num1, str1)
```

```
start()
```

Two parameters (a number and a string) are passed into the call for 'fun2()' but in the definition of the function it's expected that both parameters are numeric.

Two numeric parameters are passed into the call for 'fun1()' but only one parameter is listed in the definition for function 'fun1()'




Yet Another Common Mistake: Not Declaring Parameters

You wouldn't do it this way with pre-created functions:

```
def start():  
    print(num)
```

What is 'num'? It
has not been
declared in function
'start()'



So why do it this way with functions that you define yourself:


Etc. (Assume fun() has been defined elsewhere in the program)

start

```
def start():  
    fun(num)
```

```
start()
```

What is 'num'? It
has not been
created in function
'start()'



start (correct)

```
def start():  
    num = <Create first>  
    fun(num)
```

```
start()
```

New Problem: Results That Are Derived In One Function Only Exist While The Function Runs

Stored locally
interest = 50

```
def calculateInterest(principle, rate, time):  
    interest = principle * rate * time
```

start

```
principle = 100
```

```
rate = 0.1
```

```
time = 5
```

```
calculateInterest (principle, rate, time)
```

```
print("Interest earned $", interest)
```

Problem:

Value stored in
interest cannot be
accessed here

Solution: Have The Function Return Values Back To The Caller

```
def calculateInterest(principle, rate, time):  
    interest = principle * rate * time  
    return(interest)
```

Variable
'interest' is still
local to the
function.

```
# start  
    principle = 100  
    rate = 0.1  
    time = 5  
    interest = calculateInterest(principle,  
                                rate, time)  
    print ("Interest earned $", interest)
```

The value stored in the
variable 'interest' local
to 'calculateInterest()' is
passed back and stored in a
variable that is local to the
"start function".

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(principle, rate, time)`

Function call

`principle, rate, time = getInputs()`

Return And The End Of A Function

- A function will immediately end and return back to the caller if:
 1. A return statement is encountered (return can be empty "None")

```
def convert(catAge):  
    if (catAge < 0):  
        print("Can't convert negative age to human years")  
        return()      # Explicit return to caller (return  
                        # statement)  
    else:  
        :      :
```

2. There are no more statements in the function.

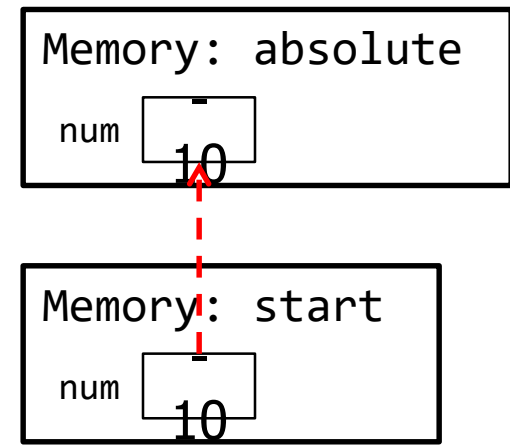
```
def introduction():  
    print()  
    print("TAMCO INC. Investment simulation program")  
    print("All rights reserved")  
    print() # Implicit return to caller (last statement)
```

Parameter Passing Vs. Return Values

- Parameter passing is used to pass information INTO a function.
 - Parameters *are copied into variables* that are local to the function.

```
def absolute(num):  
    etc.
```

```
def start():  
    num = int(input("Enter number: "))  
    absolute(num)
```



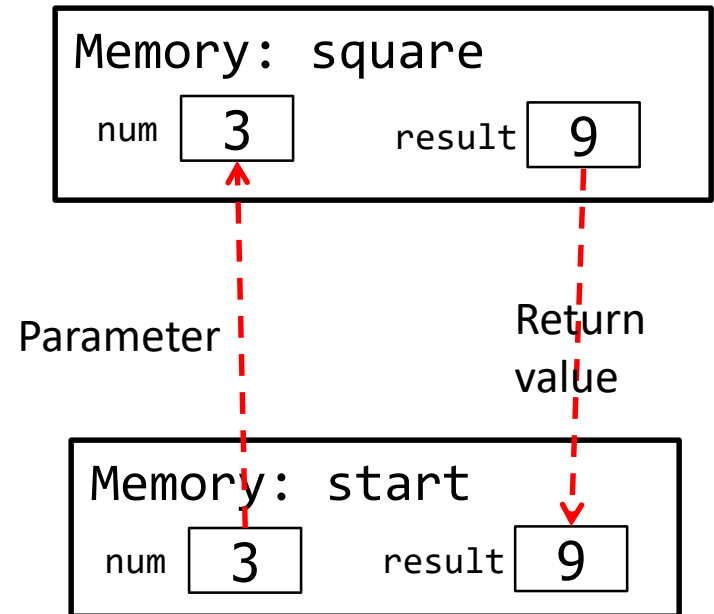
Parameter Passing Vs. Return Values

- Return values are used to communicate information OUT OF a function.
 - The return value must be stored in the caller of the function.

```
def square(num):  
    result = num * num  
    return(result)
```

```
def start():  
    num = int(input("Enter number: "))  
    result = square(num)  
    print(result)
```

```
start()
```



Good Style: Functions

1. Each function should have one well defined task. If it doesn't then this may be a sign that the function should be decomposed into multiple sub-functions.
 - a) Clear function: A function that squares a number.
 - b) Ambiguous function: A function that calculates the square and the cube of a number.
 - Writing a function that is too specific makes it less useful (in this case what if we wanted to perform one operation but not the other).
- Also functions that perform multiple tasks can be harder to test.

Good Style: Functions (2)

2. (Related to the previous point). Functions should have a self descriptive action-oriented name (verb/action phrase or take the form of a question – the latter for functions that check if something is true): the name of the function should provide a clear indication to the reader what task is performed by the function.
 - a) Good: `drawShape()`, `toUpper()`
`isNum()`, `isUpper()` # Boolean functions: ask questions
 - a) Bad: `doIt()`, `go()`, `a()`

Good Style: Functions (2)

3. Try to avoid writing functions that are longer than one screen in length.
 - a) Tracing functions that span multiple screens is more difficult.
4. The conventions for naming variables should also be applied in the naming of functions.
 - a) Lower case characters only.
 - b) With functions that are named using multiple words capitalize the first letter of each word except the first (so called “camel case”) - most common approach or use the underscore (less common). Example:
toUpper()

After This Section You Should Now Know

- How and why the top down approach can be used to decompose problems
 - What is procedural programming
- How to write the definition for a function
- How to write a function call
- How to pass information to functions via parameters
- How and why to return values from a function
- What is the difference between a local and a global variable.
- Common problems and mistakes

Thank you very much!