Functions: Decomposition And Code Reuse (Part I)

This section of notes shows you how to write functions that can be used to: decompose large problems, and to reduce program size by creating reusable sections.

Tip For Success: Reminder

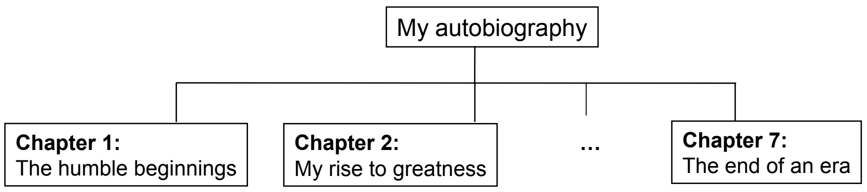
- Look through the examples and notes before class.
- This is especially important for this section because the execution of these programs will not be in sequential order.
- Instead execution will appear to 'jump around' so it will be harder to understand the concepts and follow the examples illustrating those concepts if you don't do a little preparatory work.

Solving Larger Problems

- Sometimes you will have to write a program for a large and/or complex problem.
- One technique to employ 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 score years ago with a log-shaped work station…

Breaking A Large Problem Down

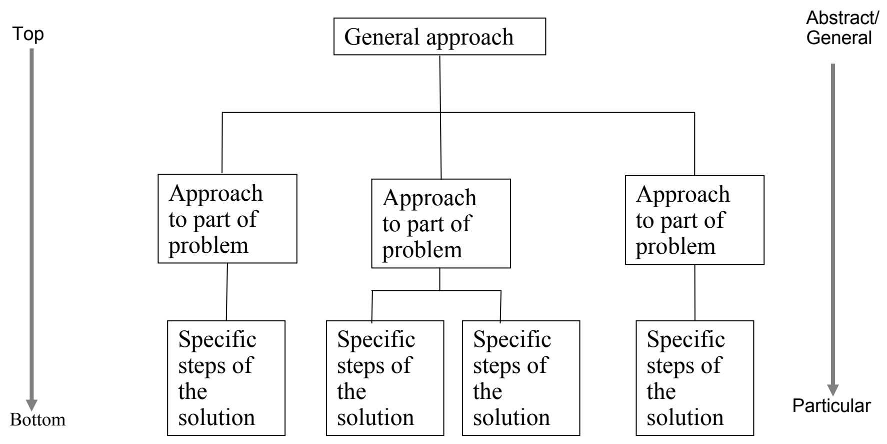
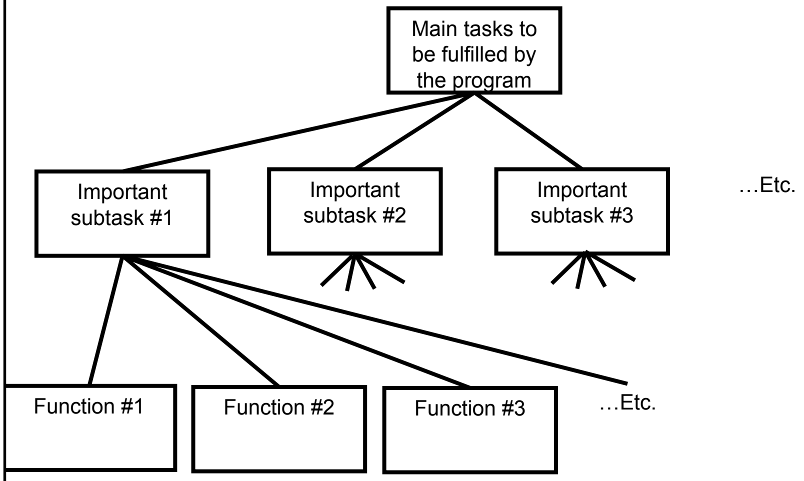

Top
Bottom
General approach
Approach to part of problem
Approach to part of problem
Approach to part of problem
Specific steps of the solution
Specific steps of the solution
Specific steps of the solution
Specific steps of the solution
Abstract/ General
Particular

Figure extracted from Computer Science Illuminated by Dale N. and Lewis J.

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” or “methods” depending upon the programming language).

Procedural Programming



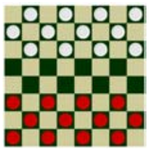
Why Decompose

- Why not just start working on the details of the solution without decomposing it into parts.
  - "*I want to *do* not plan and design!*"

Here is the first of my many witty anecdotes, it took place in a "Tim Horton's" in Balzac..

Just start writing without worrying about how things will be laid out and structured.

- Potential problems:
  - Redundancies and lack of coherence between sections.
  - Trying to implement all the details of large problem all at once may prove to be overwhelming ("*Where do I start???!!!*")



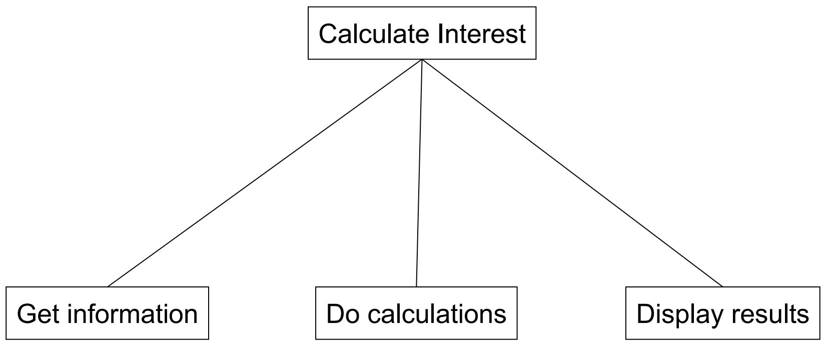
Decomposing A Problem Into Procedures

- Break down the program by what it does (described with *actions/verbs*).
- 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.
- Action/verb list:
  - Prompt
  - Calculate
  - Display

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

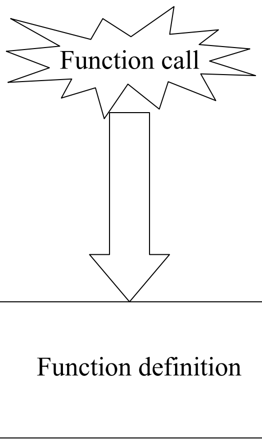


Things Needed In Order To Use Functions

- Definition
  - Instructions that indicate what the function will do when it runs.
- Call
  - Actually running (executing) the function.

- Note: a function can be called multiple (or zero) times but it can only be defined once. Why?

Functions (Basic Case)



Defining A Function

- **Format:**

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

- **Example:**

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

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

The rule in Python for specifying what statements are part of the body is to use indentation.

Calling A Function

- **Format:**

  <function name> ()

- **Example**:

  displayInstructions ()

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

- The name of the online program is “firstExampleFunction.py”

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



# Main body of code (starting execution point)
displayInstructions()
print "End of program"
```

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

- The name of the online program is “firstExampleFunction.py”

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

Function definition

```
# Main body of code (starting execution point)
displayInstructions()
print "End of program"
```

Function call

Defining The Main Body Of Code As A Function

- Rather than defining instructions outside of a function the main starting execution point can also be defined explicitly as a function.
- (The previous program rewritten to include an explicit main function) “firstExampleFunction2.py”

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

def main ():
    displayInstructions()
    print "End of program“
```

- **Important:** If you explicitly define the main function then do not forgot to explicitly call it!

```
main ()
```

Functions Should Be Defined Before They Can Be Called!

•**Correct** ☺

```
def fun ():
    print "Works"
```

Function definition

```
# main
fun ()
```

Function call

•**Incorrect** ☹

```
fun ()
```

Function call

```
def fun ():
    print "Doesn't work"
```

Function definition

Another Common Mistake

•Forgetting the brackets during the function call:

```
def fun ():
    print "In fun"

# Main function
print "In main"
fun
```

Another Common Mistake

- Forgetting the brackets during the function call:

```
def fun ():
    print “In fun”

# Main function
print “In main”
fun ()
```

The missing set of brackets does not produce a translation error

Another Common Problem: Indentation

- Recall: In Python indentation indicates that statements are part of the body of a function.
- (In other programming languages the indentation is not a mandatory part of the language but indenting is considered good style because it makes the program easier to read).
- Forgetting to indent:

```
def main ():
print "main"

main ()
```

- Inconsistent indentation:

```
def main ():
  print "first"
    print "second"

main ()
```

Yet Another Problem: Creating 'Empty' Functions

```
def fun ():



# Main
fun()
```

Problem: This statement appears to be a part of the body of the function but it is not indented???!!!

Yet Another Problem: Creating 'Empty' Functions (2)

```
def fun ():
    print



# Main
fun()
```

A function must have at least one statement

Alternative (writing an empty function: literally does nothing)

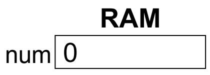
```
def fun ():
    pass

# Main
fun ()
```

What You Know: Declaring Variables

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

num = 0



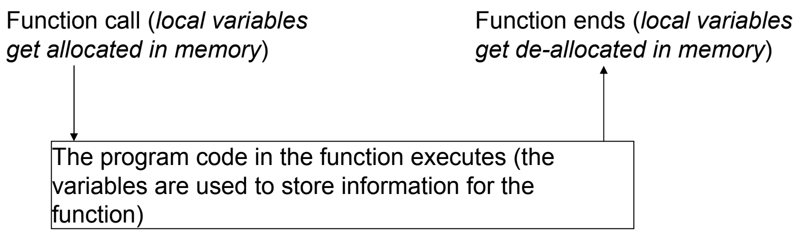
• Each variable uses up a portion of memory, if the program is large then many variables may have to be declared (a lot of memory may have to be allocated to store the contents of variables).

What You Will Learn: Using Variables That Are Local To A Function

• To minimize the amount of memory that is used to store the contents of variables only declare variables when they are needed.

• When the memory for a variable is no longer needed it can be 'freed up' and reused.

• To set up your 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.


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

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

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

Where To Create Local Variables

```
def <function name> ():
    Somewhere within
    the body of the
    function (indented
    part)
```

Example:

```
def fun ():
    num1 = 1
    num2 = 2
```

Working With Local Variables: Putting It All Together

- The full online version of this program is called "secondExampleFunction.py"

```
def fun ():
    num1 = 1
    num2 = 2
    print num1, " ", num2

# Main function
fun()
```

Working With Local Variables: Putting It All Together

- The full online version of this program is called "secondExampleFunction.py"

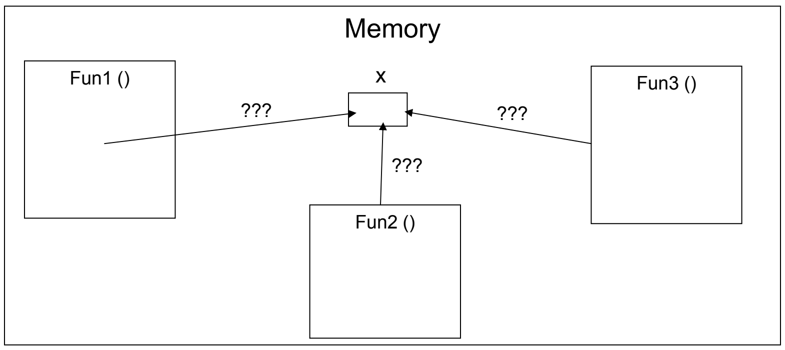
```
def fun ():
    num1 = 1
    num2 = 2
    print num1, " ", num2

# Main function
fun()
```

Variables that are local to function fun

Another Reason For Creating Local Variables

- To minimize side effects (unexpected changes that have occurred to variables after a function has ended e.g., a variable storing the age of the user takes on a negative value).
- To picture the potential problem imagine if all variables could be accessed anywhere in the program (not local).



New Problem: Local Variables Only Exist Inside A Function

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

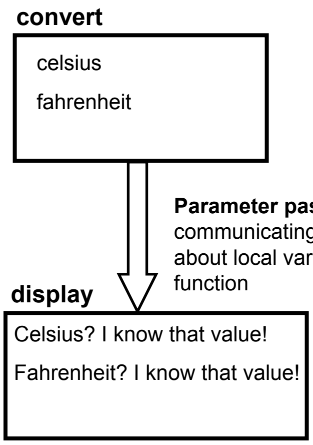
What is 'celsius'???
What is 'fahrenheit'???

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

Variables celsius and fahrenheit are local to function 'convert'

Solution: Parameter Passing

•Variables exist only inside the memory of a function:



Parameter passing: communicating information about local variables into a function

Parameter Passing (Function Definition)

- **Format:**
  def *<function name>* (*<parameter 1>*, *<parameter 2>*...):

- **Example:**
  def display (celsius, fahrenheit):

Parameter Passing (Function Call)

- **Format:**
  <function name> (*<parameter 1>*, *<parameter 2>*...)

- **Example:**
  display (celsius, fahrenheit):

Memory And Parameter Passing

•Parameters passed into functions become variables that in the local memory of that function.

Parameter num1: local to fun

num2: local to fun

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

def main ():
    num1 = 1
    fun (num1)

main ()
```

num1: local to main

Parameter Passing: Putting It All Together

•The full online version of this program is called "temperature.py"

```
def introduction ():
    print """
Celsius to Fahrenheit converter
-------------------------------
This program will convert a given Celsius temperature to an equivalent
Fahrenheit value.

    """
```

Parameter Passing: Putting It All Together (2)

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


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


# Main function
def main ():
    introduction ()
    convert ()

main ()
```

The Type And Number Of Parameters Must Match!

•Correct ☺:

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

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

# main
def main  ():
    num1 = 1
    num2 = 2
    str1 = "hello"
    fun1 (num1, num2)
    fun2 (num1, str1)

main ()
```

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'

Another Common Mistake: The Parameters Don't Match

•**Incorrect ☹:**

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

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

# main
def main ():
    num1 = 1
    num2 = 2
    str1 = "hello"
    fun1 (num1, num2)
    fun2 (num1, str1)

main ()
```

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'

Good Style: Functions

1. Each function should have one well defined task. If it doesn't then it may be a sign that it should be decomposed into multiple sub-functions.
   a) Clear function: A function that converts lower case input to capitals.
   b) Ambiguous function: A function that prompts for a string and then converts that string to upper case.
2. (Related to the previous point). Functions should have a self descriptive name: the name of the function should provide a clear indication to the reader what task is performed by the function.
   a) Good: isNum, isUpper, toUpper
   b) Bad: doIt, go
3. Try to avoid writing functions that are longer than one screen in size.
   a) Tracing functions that span multiple screens is more difficult.

Good Style: Functions (2)

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 but the first (most common approach) or use the underscore (less common).

Why Employ Problem Decomposition And Modular Design

- Drawback
  - Complexity – understanding and setting up inter-function communication may appear daunting at first.
  - Tracing the program may appear harder as execution appears to "jump" around between functions.
- Benefit
  - Solution is easier to visualize and create (only one part of a time).
  - Easier to test the program (testing all at once increases complexity).
  - Easier to maintain (if functions are independent changes in one function can have a minimal impact on other functions, if the code for a function is used multiple times then updates only have to be made once).
  - Less redundancy, smaller program size (especially if the function is used many times throughout the program).

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 and why to declare variables locally
- How to pass information to functions via parameters
- Good programming principles for implementing functions