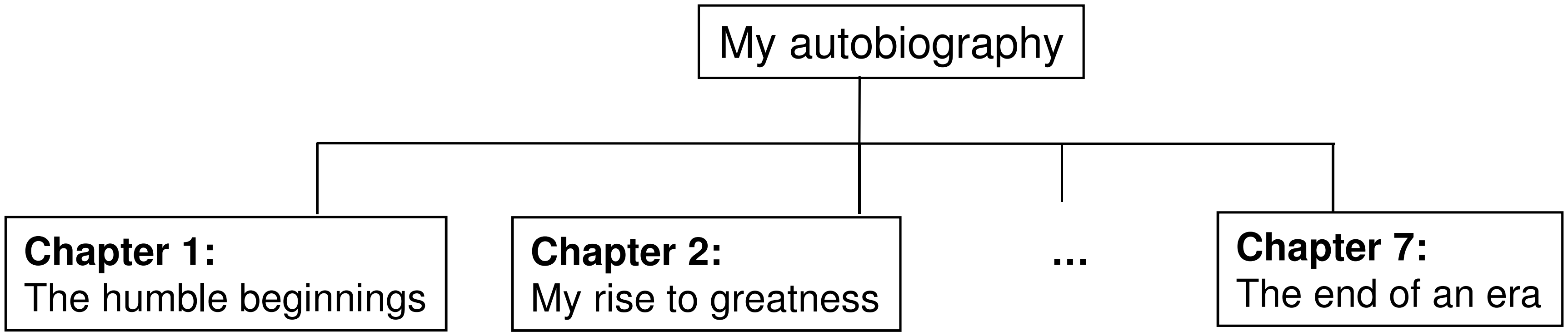
Functions: Decomposition And Code Reuse

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.

Top Down Design

1. Start by outlining the major parts (structure)


My autobiography
Chapter 1: The humble beginnings
Chapter 2: My rise to greatness
…
Chapter 7: The end of an era

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

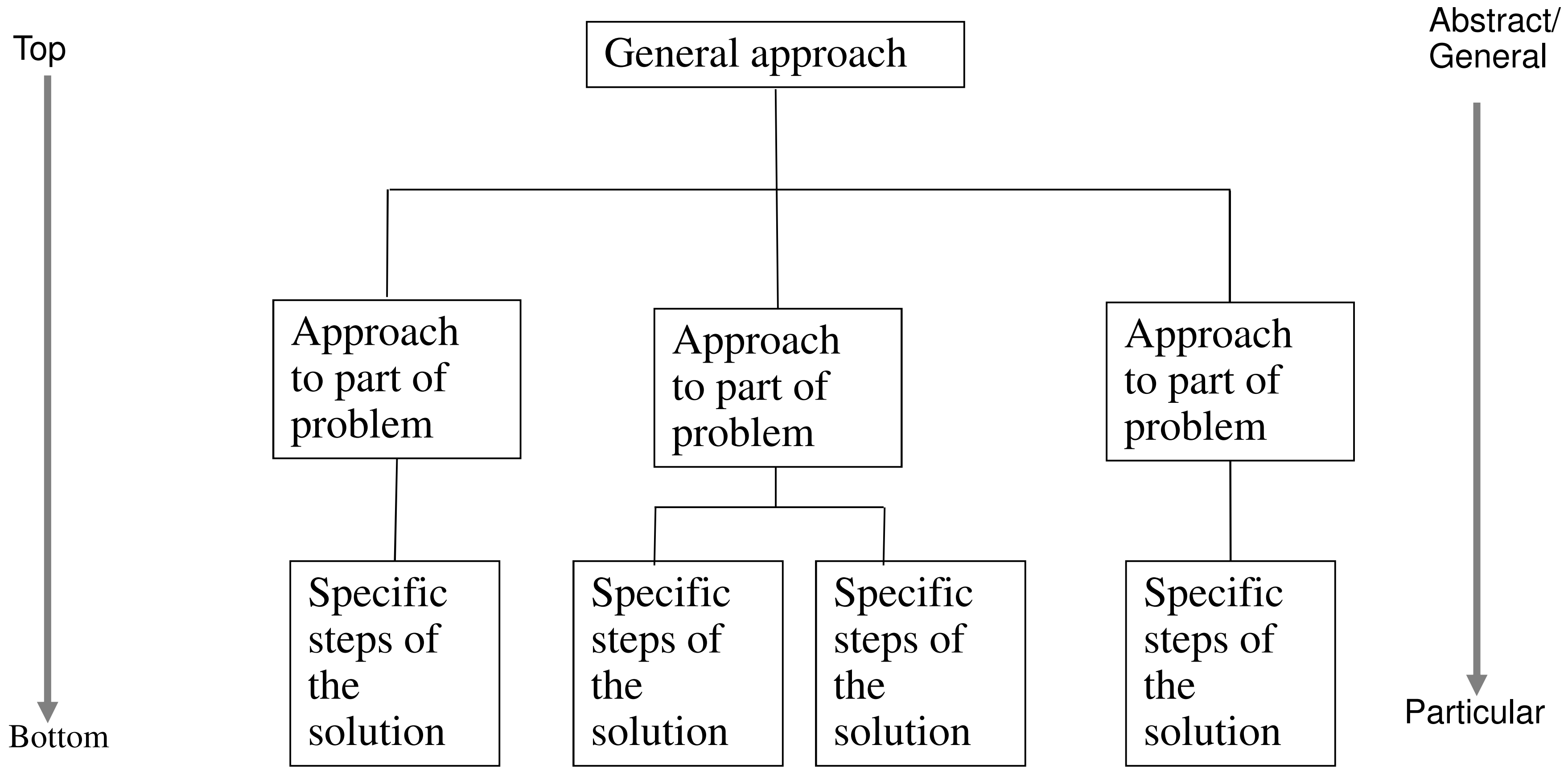
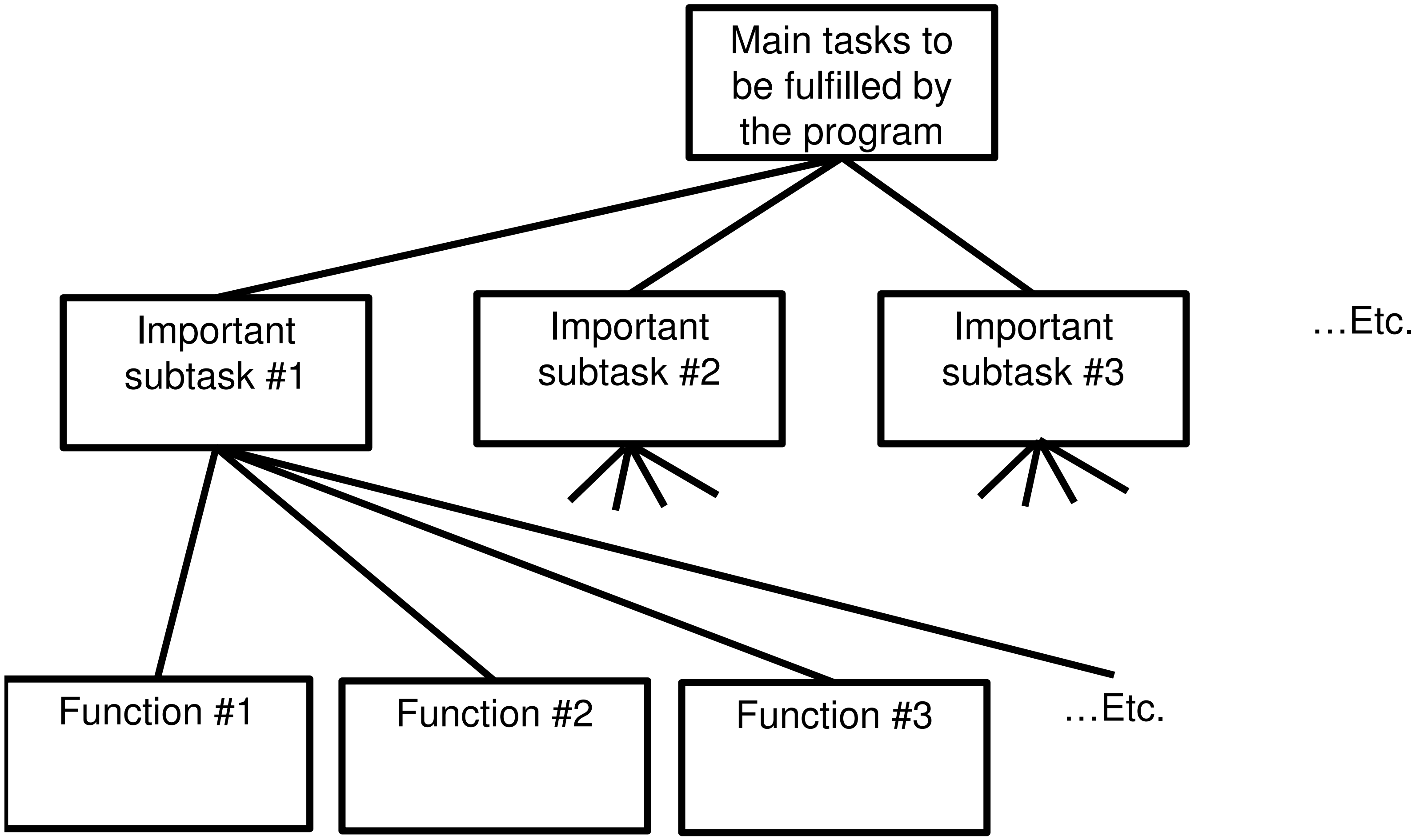

Top
Bottom
General approach
Approach to part of problem
Approach to part of problem
Approach to part of problem
Specific steps of the solution
Specific steps of the solution
Specific steps of the solution
Specific steps of the solution
Abstract/ General
Particular

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



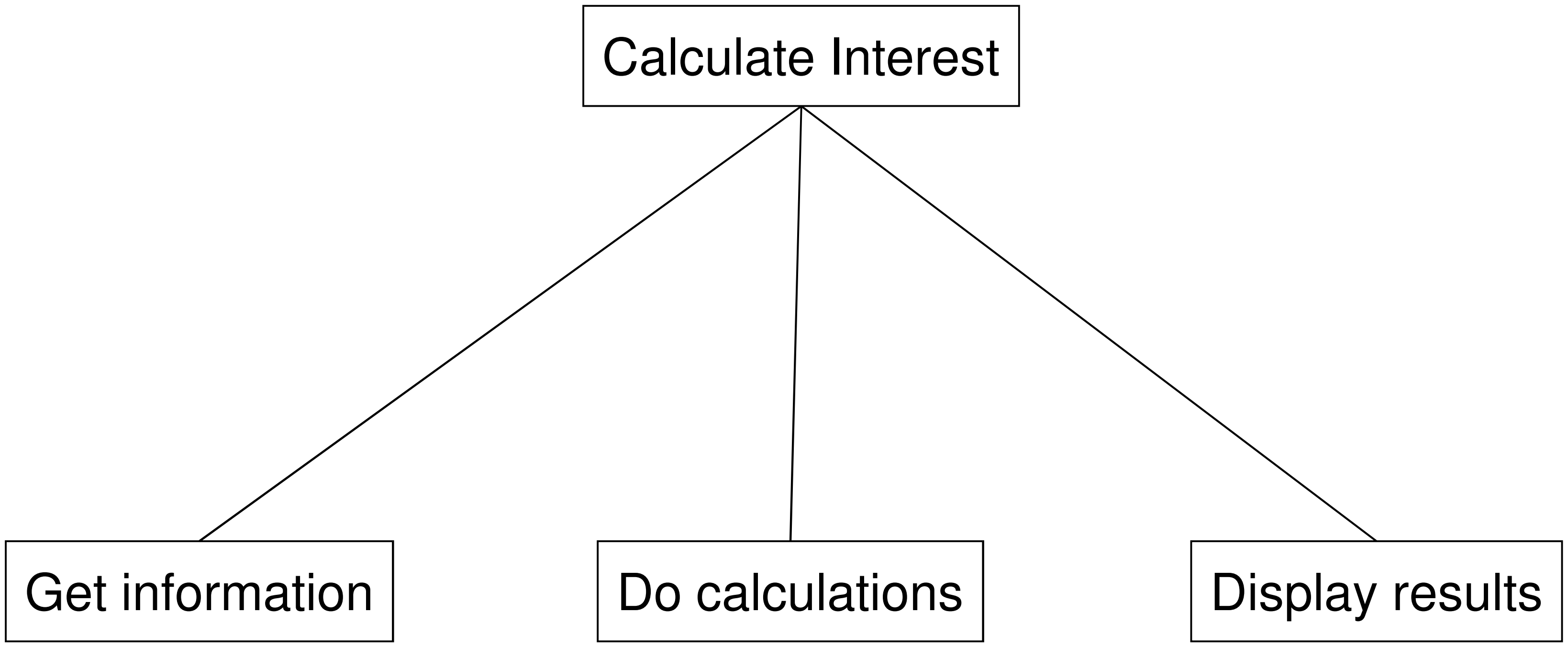
Decomposing A Problem Into Procedures

- 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?

Defining A Function

- **Format:**

```
def <function name> ():
    body[1]
```

- **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

- Name of the example program: firstExampleFunction.py

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

Function definition

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

Function call

Functions Should Be Defined Before They Can Be Called!

- **Correct ☺**

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

$\underbrace{}_{}$ **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 ()
```

Another Common Problem: Indentation (2)

- Inconsistent indentation:

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

main ()
```

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

```
def fun ():
   print ()


# Main
fun()
```

A function must have at least one statement

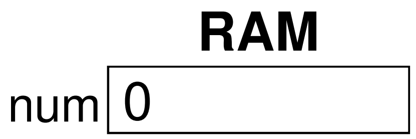
Alternative (writing an empty function: literally does nothing)

```
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

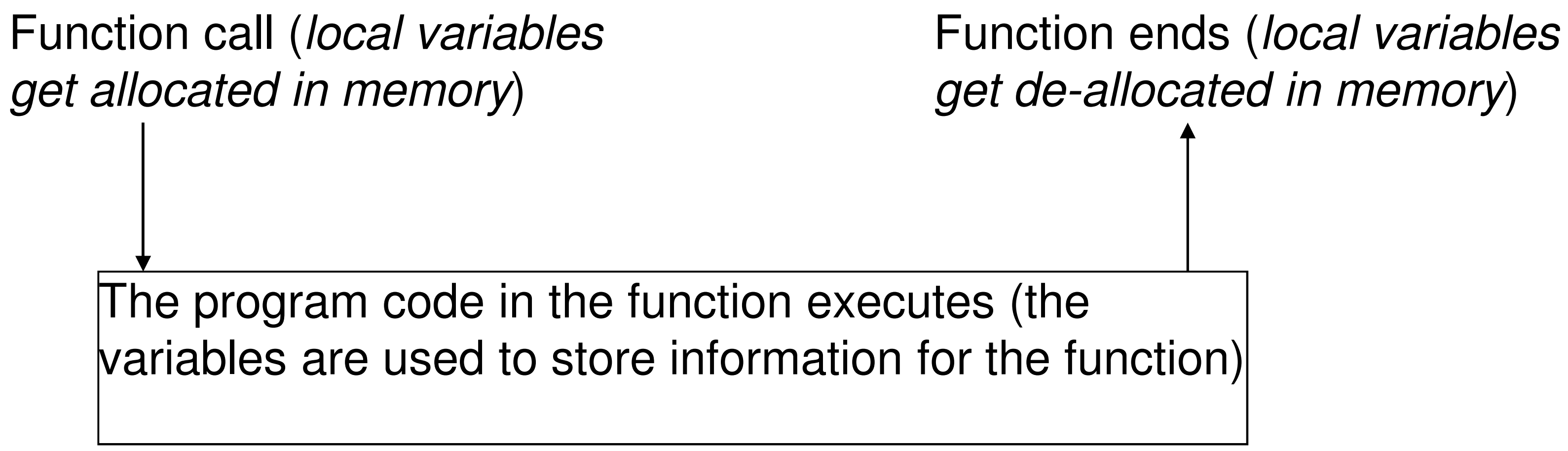


- Each variable uses up a portion of memory, if the program is large then many variables may have to be declared (a lot of memory may have to be allocated to store the contents of variables).

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.

What You Will Learn: Using Variables That Are Local To A Function (2)



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

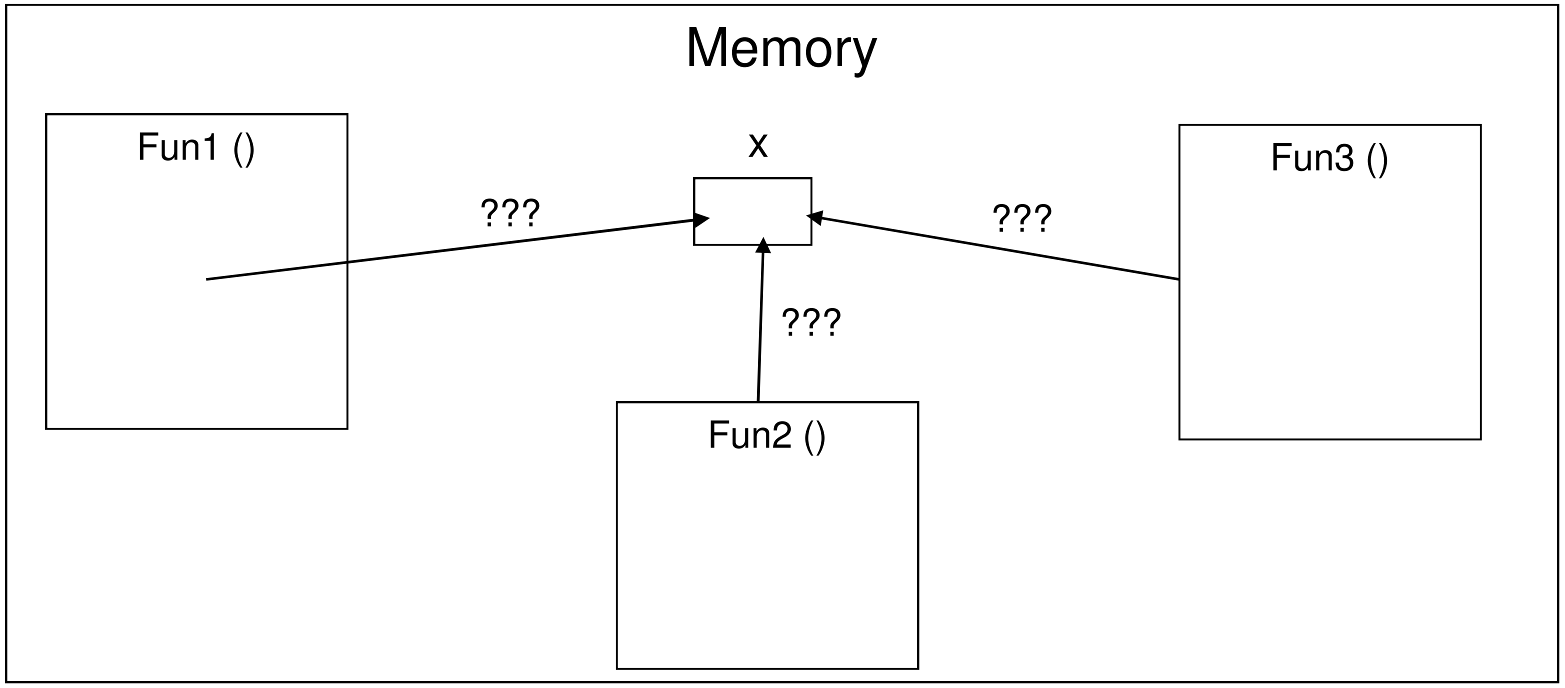
•Name of the example program: secondExampleFunction.py

```
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

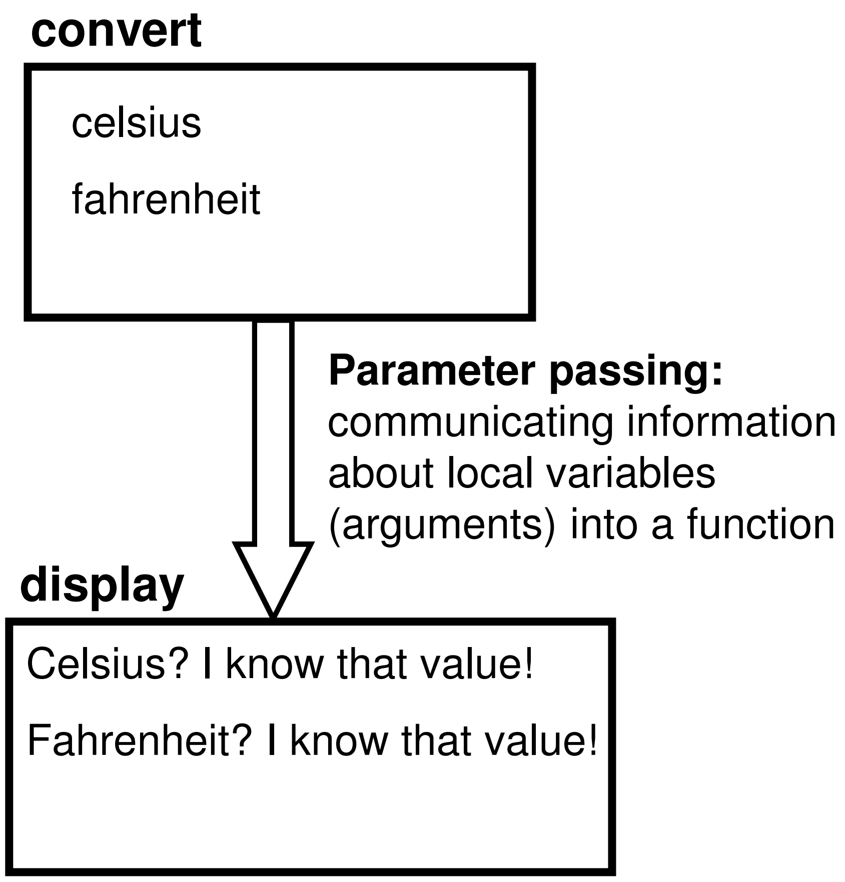
```
def display ():
    print ("“)
    print ("Celsius value: ", celsius)
    print ("Fahrenheit value :", fahrenheit)

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

What is ‘celsius’???
What is ‘fahrenheit’???

Variables celsius and fahrenheit are local to function ‘convert’

Solution: Parameter Passing

- Variables exist only inside the memory of a function:


convert
celsius
fahrenheit
Parameter passing:
communicating information
about local variables
(arguments) into a function
display
Celsius? I know that value!
Fahrenheit? I know that value!

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 as arguments into functions become variables 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)

num1 = 1
fun (num1)
```

num1: local to main

Parameter Passing: Putting It All Together

•Name of the example program: 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 = float(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'

Default Parameters

- Can be used to give function arguments some default values if none are provided.
- Example function definition:

```
def fun (x = 1, y = 1):
   print (x, y)
```

- Example function calls (both work):

```
- fun ()
- fun (2, 20)
```

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.

<u>Good Style: Functions (2)</u>

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).

Parameter Passing

- What you know about scope: Parameters are used to pass the contents of variable into functions (because the variable is not in scope).

```
def fun1 ():
    num = 10
    fun2 (num)


def fun2 (num):
    print num
```

Scope

- The scope of an identifier (variable, constant) is where it may be accessed and used.
- In Python[1]:
  - An identifier comes into scope (becomes visible to the program and can be used) after it has been declared.
  - An identifier goes out of scope (no longer visible so it can no longer be used) at the end of the indented block where the identifier has been declared.

1 The concept of scoping applies to all programming languages. The rules for determining when identifiers come into and go out of scope will vary.

Scope: An Example

Scope of num

```
def fun1 ():
    num = 10
    # statement
    # statement
    # End of fun1
```

‘num’ comes into scope (is visible and can be used)

(End of function): num goes out of scope, no longer accessible

Num is no longer in scope

```
def fun2 ():
    print num
    :      :
```

Error: num is an unknown identifier

Scope: A Variant Example

```
def fun1 ():
    num = 10
    # statement
    # statement
    # End of fun1


def fun2 ():
    fun1 ()
    num = 20
    :      :
```

What happens at this point?

Why?

Global Scope

- Identifiers (constants or variables) that are declared within the body of a function have a local scope (the function).

```
def fun ():
    num = 12
    # End of function fun
```

Scope of num is the function

- Identifiers (constants or variables) that are declare outside the body of a function have a global scope (the program).

```
num = 12
def fun1 ():
  # Instruction

def fun2 ():
  # Instruction

# End of program
```

Scope of num is the entire program

Global Scope: An Example

- Name of the example program: globalExample1.py

```
num1 = 10

def fun ():
   print (num1)

def main ():
   fun ()
   print (num2)

num2 = 20

main ()
```

Global Variables: General Characteristics

- You can access the contents of global variables anywhere in the program.
- In most programming languages you can also modify global variables anywhere as well.
  - This is why the usage of global variables is regarded as bad programming style, they can be accidentally modified anywhere in the program.
  - Changes in one part of the program can introduce unexpected side effects in another part of the program.
  - So unless you have a compelling reason you should NOT be using global variables but instead you should pass values as parameters.

Global Variables: Python Specific Characteristic

- Name of the example program: globalExample2.py

```
num = 1

def fun ():
   num = 2
   print (num)

def main ():
   print (num)
   fun ()
   print (num)


main ()
```

Python Globals: Read But Not Write Access

- By default global variables can be accessed globally (read access).
- Attempting to change the value of global variable will only create a new local variable by the same name (no write access).

Global num

```
num = 1
def fun ():
```

Local num

```
   num = 2
   print (num)
```

- Prefacing the name of a variable with the keyword ‘global’ will indicate that all references in that function will then refer to the global variable rather than creating a local one.

  global <*variable name*>

Globals: Another Example

- Name of the example program: globalExample3.py

```
num = 1

def fun1 ():
    num = 2
    print (num)

def fun2 ():
    global num
    num = 2
    print (num)
```

Globals: Another Example (2)

```
def main ():
   print (num)
   fun1 ()
   print (num)
   fun2 ()
   print (num)


main ()
```

Function Pre-Conditions

- Specifies things that must be true when a function is called.
- Examples:

```
# Precondition: Age must be a non-negative number
def convertCatAge (catAge):
    humanAge = catAge * 7
    return humanAge

# Precondition: y is a numeric non-zero value
def divide (x, y):
     z = x / y
    return z
```

Function Post-Conditions

- Specifies things that must be true when a function ends.
- Example:

```
def absoluteValue (number):
    if (number < 0):
        number = number  * -1
    return number
    # Post condition: number is non-negative
```

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 (decompose the problem so only one part of a time must be dealt with).
  - 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).
  - Smaller programs size: if the function is called many times rather than repeating the same code, the function need only be defined once and then can be called many times.