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.

Example Programs

- Location (via the WWW):
  - http://pages.cpsc.ucalgary.ca/~tamj/217/examples/decomposition
- Location (via the CPSC UNIX network):
  - /home/courses/217/examples/decomposition

Tip For Success: Reminder

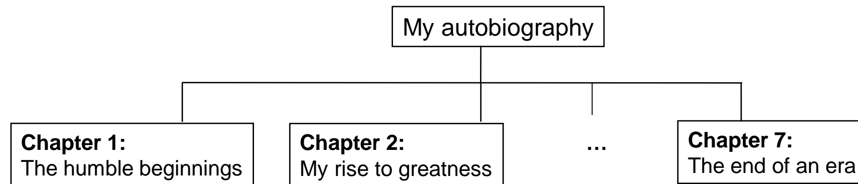
- Look through the examples and notes before class.
- This is especially important for this section because the execution of these programs will not be in sequential order.
- Instead execution will appear to ‘jump around’ so it will be harder to understand the concepts and follow the examples illustrating 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 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

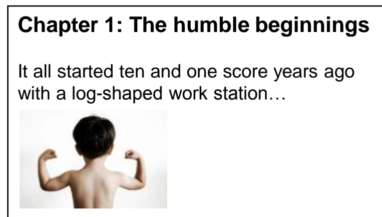
1. Start by outlining the major parts (structure)



2. Then implement the solution for each part

Chapter 1: The humble beginnings

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



Breaking A Large Problem Down

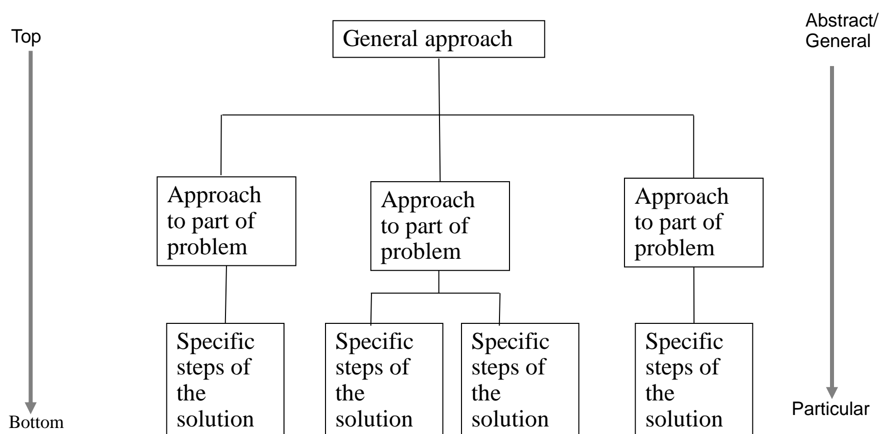
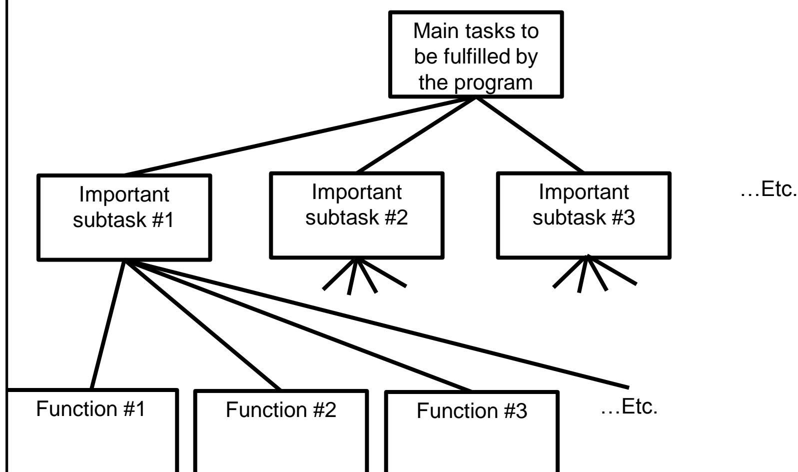


Figure extracted from Computer Science Illuminated by Dale N. and Lewis J.

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!"*

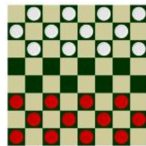
Here is the first of my many witty anecdotes, it took place in a "Tim Horton's" in Balzac..

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???!!!*")



An actual assignment from this class

Decomposing A Problem Into Procedures

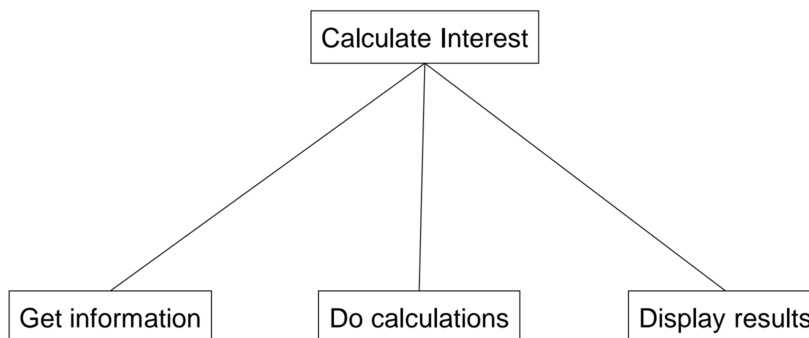
•Break down the program by what it does (described with *actions/verbs*).

•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)



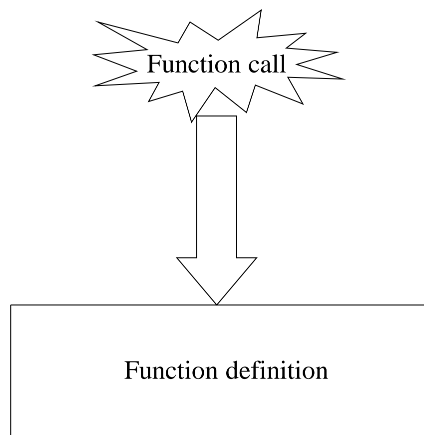
Things Needed In Order To Use Functions

- 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 on how to use the program")
```

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:**

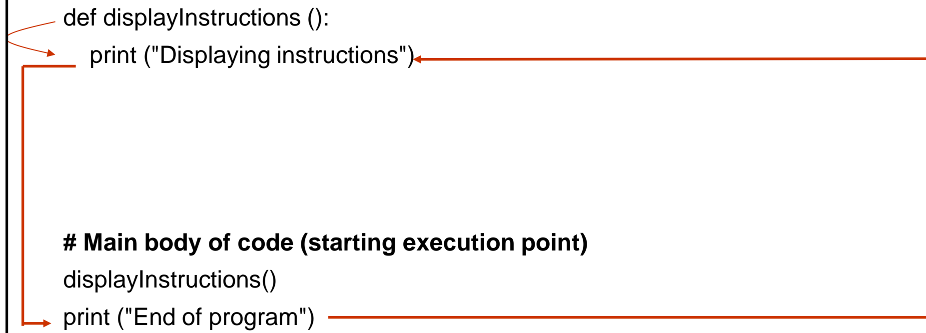
```
<function name> ()
```

- **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")


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

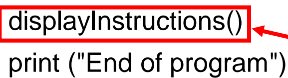
Functions: An Example That Puts Together All The Parts Of The Easiest Case

•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

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 start function) "firstExampleFunction2.py"

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

def start ():
    displayInstructions()
    print ("End of program")
```

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

```
start ()
```

Don't forget to start your program!

Functions Should Be Defined Before They Can Be Called!

- **Correct ☺**

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

Function definition

```
# start
fun ()
```

Function call

- **Incorrect ☹**

```
fun ()
```

Function call

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

Function definition

Another Common Mistake

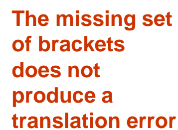
•Forgetting the brackets during the function call:

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

# start function
print ("In start")
fun
```

Another Common Mistake

•Forgetting the brackets during the function call:

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

# start function
print ("In start")
fun ()
```



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 start ():
print ("start“)

start ()
```

Another Common Problem: Indentation (2)

- Inconsistent indentation:

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

start ()
```

Yet Another Problem: Creating 'Empty' Functions

```
def fun ():



# start
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 ()



# start
fun()
```

A function must have at least one statement

Alternative (writing an empty function: literally does nothing)

```
def fun ():
    pass

# start
fun ()
```

What You Know: Declaring Variables

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

RAM

num = 0   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 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.

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

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

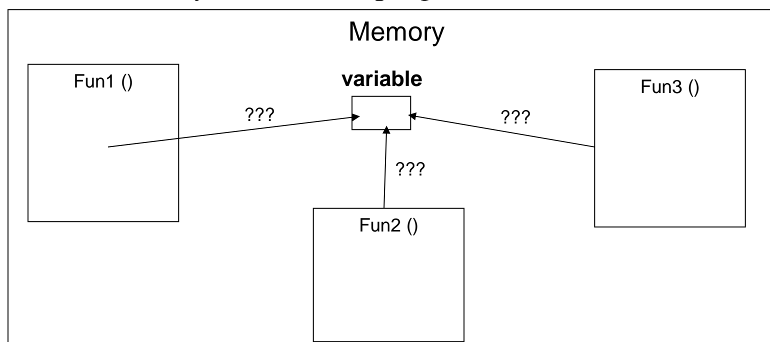
```
def fun ():
    num1 = 1
    num2 = 2
    print (num1, " ", num2)

# start 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 visualize 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'

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

Solution: Parameter Passing

•Variables only exist 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.

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

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

start ()
```

Parameter num1: local to fun

num2: local to fun

num1: local to start

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)


# start function
def start ():
    introduction ()
    convert ()

start ()
```

Parameter Passing: Important Recap!

- A parameter is copied into a local memory space.

```
display (celsius, fahrenheit)    # Function call

  Make copy         Make copy

  def display (celsius, fahrenheit):    # Function definition
```

- For stylistic reasons: the names should match (unless there is a compelling reason not to match).
- However because they are separate memory locations there are *no technical reasons* why the parameter and the local variable names must match.

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'

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)

# 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:

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

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

So why do it this way:

Etc. (Assume fun has been defined)

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

start ()
```

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

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)

Default Parameters: Application

- It can be useful if one function may require different parameters at different times.
- Examples: print(), input()
- Use of the print function does not require other programmers to remember different versions of these functions
- (The print() function isn't defined this way – fortunately)
  - e.g., printInt(10), printFloat(10.2), printIntFloat(10, 9.9)
- (Print is written to automatically allow for different parameters)
  - E.g., print(10), print(3.33), print(3.14,10) etc..
- Providing default parameters could allow for functions like these to be called using different parameter lists but not requiring different versions of the functions to be defined/called.

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 user for a string and then converts that string to upper case.
2. (Related to the previous point). Functions should have a self descriptive action-oriented name (verb or a question): 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.

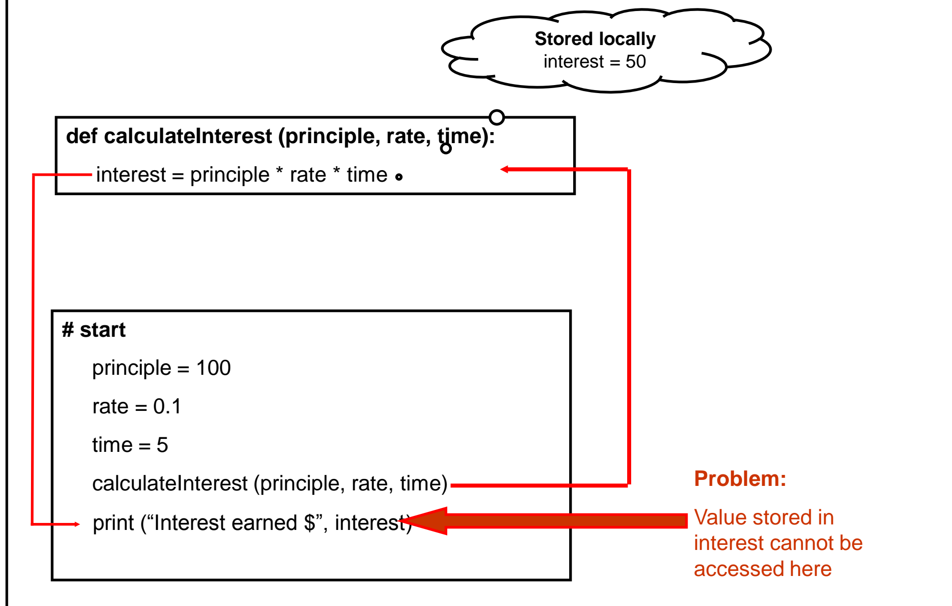
Good Style: Functions (2)

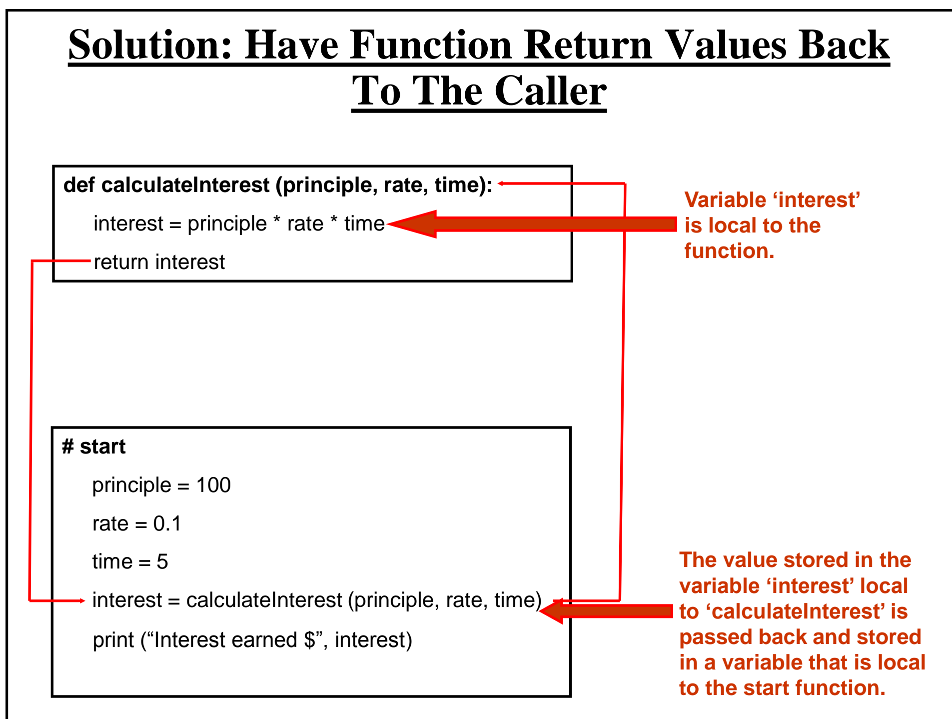
4. The conventions for naming variables should also be applied in the naming of functions.
   a) Lower case characters only.
   b) With functions that are named using multiple words capitalize the first letter of each word but the first (most common approach) or use the underscore (less common). Example: toUpper()

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)
```


New Problem: Results That Are Derived In One Function Only Exist Until The Function Ends
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 Function Return Values Back To The Caller
def calculateInterest (principle, rate, time):
interest = principle * rate * time
return interest
Variable 'interest' is 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 (Single value returned):

return (<*value returned*>) # Function definition

<*variable name*> = <*function name*> () # Function call

•Example (Single value returned):

return(interest) # Function definition

interest = calculateInterest (principle, rate, time) # Function call

Using Return Values

•Format (Multiple values returned):

return (<*value1*>, <*value 2*>...) # Function definition

<*variable 1*>, <variable 2>... = <*function name*> () # Function call

•Example (Multiple values returned):

return (principle, rate, time) # Function definition

principle, rate, time = getInputs (principle, rate, time) # Function call

Using Return Values: Putting It All Together

•Name of the example program: interest.py

```
def introduction ():
    print ("""
Simple interest calculator
-------------------------------
With given values for the principle, rate and time period this program
will calculate the interest accrued as well as the new amount (principle
plus interest).
    """)
```

Using Return Values: Putting It All Together (2)

```
def getInputs ():
    principle = float (input("Enter the original principle: "))
    rate = float(input("Enter the yearly interest rate %"))
    rate = rate / 100
    time = input("Enter the number of years that money will be invested:
                  ")
    time = float(time)
    return (principle, rate, time)

def calculate (principle, rate, time):
    interest = principle * rate * time
    amount = principle + interest
    return (interest, amount)
```

Using Return Values: Putting It All Together (3)

```
def display (principle, rate, time, interest, amount):
    temp = rate * 100
    print ("")
    print ("Investing $%.2f" %principle, "at a rate of %.2f" %temp, "%")
    print ("Over a period of %.0f" %time, "years...")
    print ("Interest accrued $", interest)
    print ("Amount in your account $", amount)
```

Using Return Values: Putting It All Together (4)

```
# start function
def start ():
    principle = 0
    rate = 0
    time = 0
    interest = 0
    amount = 0

    introduction ()
    principle, rate, time = getInputs ()
    interest, amount = calculate (principle, rate, time)
    display (principle, rate, time, interest, amount)

start ()
```

Yet Another Common Mistake: Not Saving Return Values

- Just because a function returns a value does not automatically mean the value will be usable by the caller of that function.

```
def fun ():
   return (1)
```

This value has to be stored or used in some expression by the caller

- That is because return values have to be explicitly saved by the caller of the function.
- Example

```
def calculateArea ():
    length = 4
    width = 3
    area = length * width
    return (area)
```

```
# start
area = 0
calculateArea()
print area
```

```
# Fixed start
area = 0
area = calculateArea ()
print area
```

Local Variables

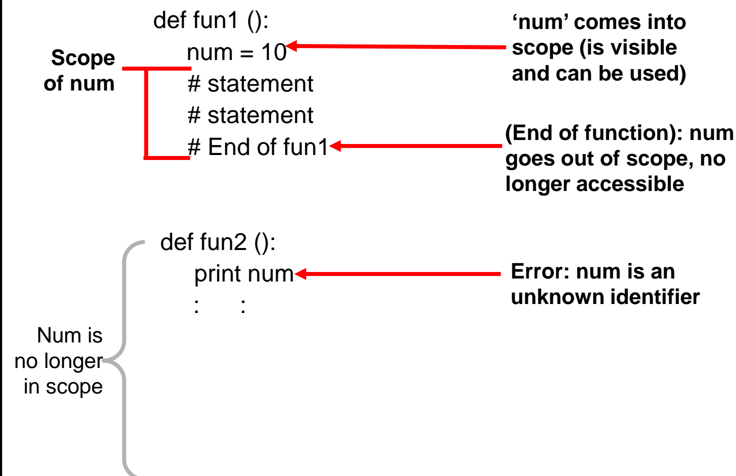
- What you know:
  - How to declare variables that only exist for the duration of a function call.
  - Why should variables be declared locally.
- What you will learn:
  - How scoping rules determine where variables can be accessed.
  - The difference between local and global scope.
  - (Previous examples may have implicitly included some of these concepts but the next section will explicitly describe how they work).

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 with a particular language.

Scope: An Example



Scope: A Variant Example

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


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

What happens at this point? (pointing to `num = 20`)

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 start ():
  fun ()
  print (num2)

num2 = 20

start ()
```

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

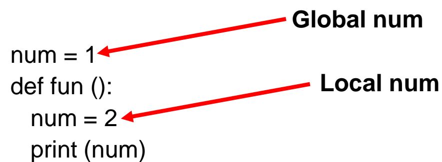
•Name of the example program: globalExample2.py

```
num = 1

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

def start ():
    print (num)
    fun ()
    print (num)


start ()
```

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).



•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 start ():
    print (num)
    fun1 ()
    print (num)
    fun2 ()
    print (num)


start ()
```

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
```

Ensuring That Preconditions Are Met

- If a function is called when the preconditions are not met will result in a serious error then the author of the function should not allow the instructions of that function to execute.
- Example (a better version of the previous example):

```
# Precondition: Age must be a non-negative number
def convertCatAge (catAge):
    if (catAge >= 0):
        humanAge = catAge * 7
    else:
        print("Age cannot be negative")
        humanAge = -1        #JT: Signal to caller than an error occurred
    return (humanAge)
```

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
```

Documenting Functions

- Functions are a 'mini' program.
- Consequently the manner in which an entire program is documented should also repeated in a similar process for each function:
  - Features list.
  - Limitations, assumptions or preconditions e.g., if a function will divide two parameters then the documentation should indicate that the function precondition is the denominator is not zero.
  - (Authorship and version number may or may not be necessary for the purposes of this class although they are frequently included in actual practice).

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.

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
- How and why to return values from a function.
- What is the difference between a local and a global variable.
- How to implement and test and program that is decomposed into functions.
- Two approaches for problem solving.