Breaking Problems Down

This section of notes shows you how to break down a large problem into smaller parts that are easier to implement and manage.

Problem Solving Approaches

Bottom up

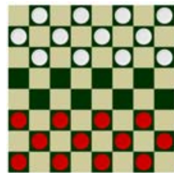
Top down

Bottom Up Approach To Design

Start implementing all details of a solution without first developing a structure or a plan.

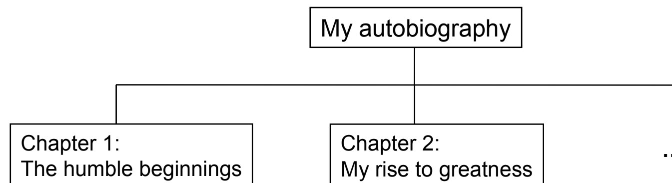
Here is the first of my many witty anecdotes, it took place in a “Tim Horton’s” in Balzac..

- Potential problems:
  - (Generic problems): Redundancies and lack of coherence between sections.
  - (Programming specific problem): Trying to implement all the details of large problem all at once may prove to be overwhelming.



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 seven and one score years ago with a log-shaped work station…

Top-Down Approach: Breaking A Large Problem Down

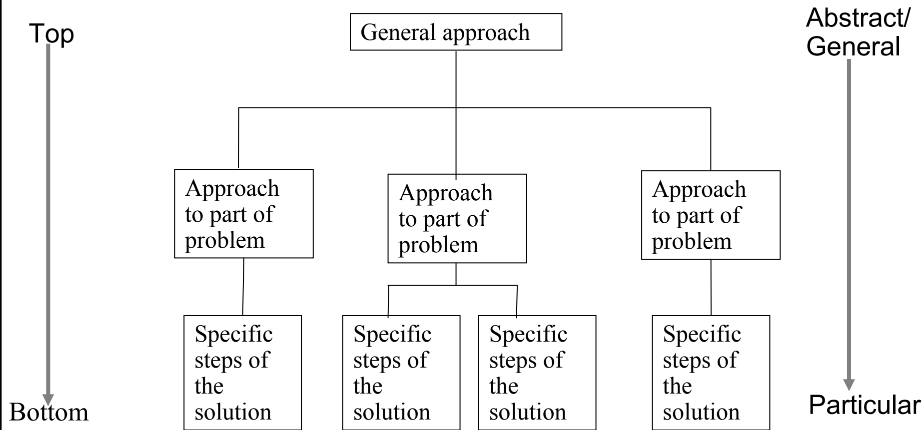
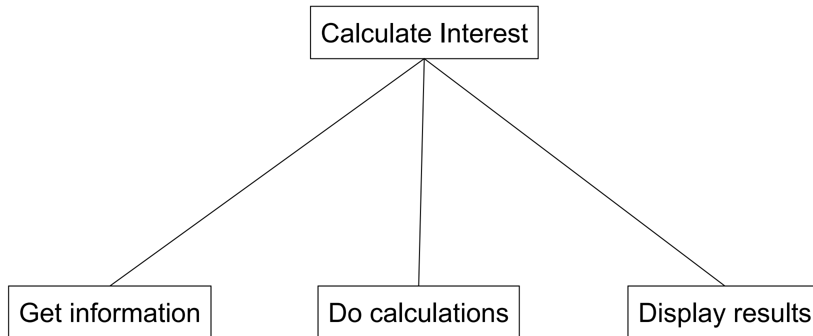


Figure extracted from Computer Science Illuminated by Dale N. and Lewis J.

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.

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

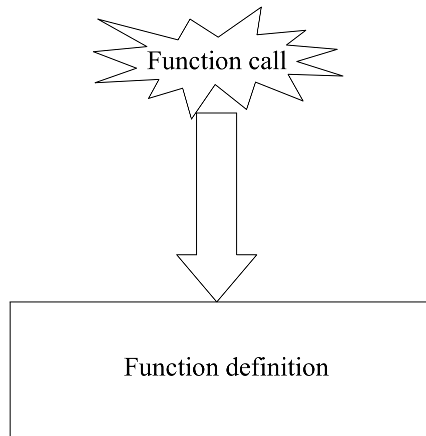


Situations In Which Functions Are Used

Definition
- Indicating what the function will do when it runs

Call
- Getting the function to run (executing the function)

Functions (Basic Case)



Defining A Function

Format:

def <*function name*> ():

body

Example:

function displayInstructions ():

print “Displaying instructions”

Calling A Function

Format:

function name ()

Example:

```
displayInstructions ()
```


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

The full version of this program can be found in UNIX under /home/217/examples/functions/firstExampleFunction.py

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

# main function
displayInstructions()
print "End of program"
```

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

The full version of this program can be found in UNIX under /home/217/examples/functions/firstExampleFunction.py

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

Function definition

```
# main function
displayInstructions()
print "End of program"
```

Function call

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
```

James Tam

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

James Tam

Do Not Create 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???!!!

Do Not Create Empty Functions

```
def fun ():
    print ""


# Main
fun()
```

A function must have at least one indented statement

What You Know: Declaring Variables

- Variables are memory locations that are used for the temporary storage of information.
- num = 0

RAM

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 – used up 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 locally 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 version of this example can be found in UNIX under
/home/courses/217/examples/functions/secondExampleFunction.py

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

# Main function
fun()
```

Working With Local Variables: Putting It All Together

The full version of this example can be found in UNIX under /home/courses/217/examples/functions/secondExampleFunction.py

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

# Main function
fun()
```

Variables that are local to function fun

Problem: Local Variables Only Exist Inside A Function

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

def convert ():
   celsius = 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’

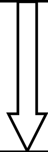
Solution: Parameter Passing

Variables exist only inside the memory of a function:

convert

celsius

fahrenheit



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):

Parameter Passing: Putting It All Together

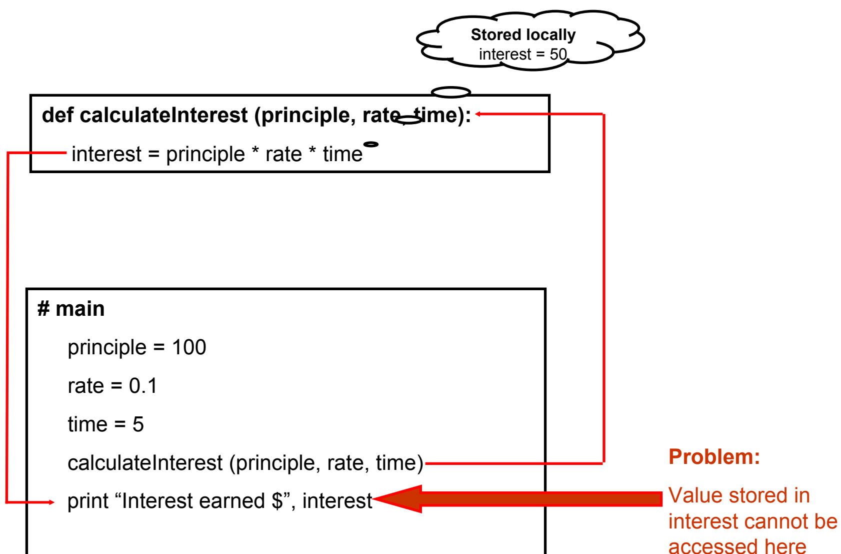
The full version of this program can be found in UNIX under /home/217/examples/functions/temperature.py

```
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
introduction ()
convert ()
```


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



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.

```
# main
    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 main 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

The full version of this program can be found in UNIX under /home/217/examples/functions/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, rate, time):
    principle = input("Enter the original principle: ")
    rate = input("Enter the yearly interest rate %")
    rate = rate / 100.0
    time = input("Enter the number of years that money will be invested: ")
    return principle, rate, time

def calculate (principle, rate, time, interest, amount):
    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 "With an investment of $", principle, " at a rate of", temp, "%",
    print " over", time, " years..."
    print "Interest accrued $", interest
    print "Amount in your account $", amount
```

Using Return Values: Putting It All Together (4)

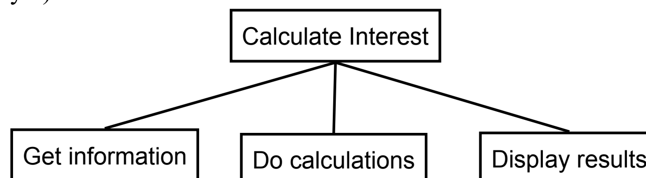
```
# Main function
principle = 0
rate = 0
time = 0
interest = 0
amount = 0

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


Testing Functions

This is an integral part of the top down approach to designing programs.

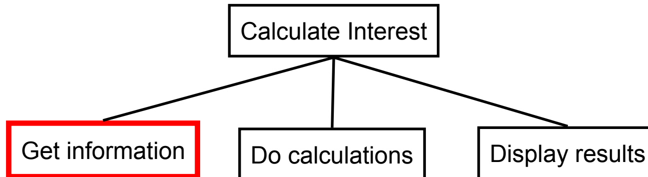
Recall with the top down approach:

1. Outline the structure of different parts of the program without implementing the details of each part (i.e., specify what functions that the program must consist of but don't write the code for the functions yet).



Testing Functions

2. Implement the body of each function, one-at-a-time.




```
# Get information
def getInput (principle, rate, time):
   principle = input (“Enter the principle: “)
   rate = input("Enter the yearly interest rate %")
   rate = rate / 100.0
   time = input("Enter the number of years that
                  money will be invested: ")
   return principle, rate, time
```


Testing Functions

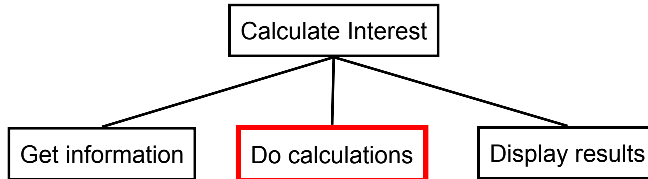
2. As each function has been written test each one to check for errors.

```
# main
   principle, rate, time = getInput()
   print “principle”, principle
   print “rate”, rate
   print “time”, time
```

```
# Get information
def getInput (principle, rate, time):
            :              :
   return principle, rate, time
```

Testing Functions

2. As each function has been written test each one to check for errors.




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


Testing Functions

2. As each function has been written test each one to check for errors.

```
# main
# Test case 1: Interest = 0, Amount = 0
interest, amount = calculate (0, 0, 0, interest, amount)
print "interest", interest, ' ', "amount", amount

# Test case 2: Interest = 50, Amount = 150
interest, amount = calculate (100, 0.1, 5, interest, amount)
print "interest", interest, ' ', "amount", amount
```

```
# Do calculations
def calculate (principle, rate, time, interest,
                    amount):
    interest = principle * rate * time
    amount = principle + interest
    return interest, amount # 0, 0
```

Testing Functions

2. As each function has been written test each one to check for errors.

```
# main
# Test case 1: Interest = 0, Amount = 0
interest, amount = calculate (0, 0, 0, interest, amount)
print "interest", interest, ' ', "amount", amount

# Test case 2: Interest = 50, Amount = 150
interest, amount = calculate (100, 0.1, 5, interest, amount)
print "interest", interest, ' ', "amount", amount
```

```
# Do calculations
def calculate (principle, rate, time, interest,
               amount):
    interest = principle * rate * time
    amount = principle + interest
    return interest, amount # 50, 100
```


The Type And Number Of Parameters Must Match!

Correct ☺:

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

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

# main
num1 = 1
num2 = 2
str1 = "hello"

fun1 (num1, num2)
fun2 (num1, str1)
```

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'

The Type And Number Of Parameters Must Match! (2)

Incorrect ☹:

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

def fun2 (num1, str1):
    num1 = str1 + 1
    print num1, str1


# main
num1 = 1
num2 = 2
str1 = "hello"

fun1 (num1, num2)
fun2 (num1, str1)
```

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'

Program Design: Finding The Candidate Functions

- The process of going from a problem description (words that describe what a program is supposed to do) to writing a program that fulfills those requirements cannot be summarized in just a series of steps that fit all scenarios.
- The first step is to look at verbs either directly in the problem description (indicates what actions should the program be capable of) or those which can be inferred from the problem description.
- Each action may be implemented as a function but complex actions may have to be decomposed further into several functions.

Program Design: An Example Problem

(Paraphrased from the book "Pascal: An introduction to the Art and Science of Programming" by Walter J. Savitch.

Problem statement:

Design a program to make change. Given an amount of money, the program will indicate how many quarters, dimes and pennies are needed. The cashier is able to determine the change needed for values of a dollar and above.

Actions that may be needed:

•Action 1: Prompting for the amount of money

•Action 2: Computing the combination of coins needed to equal this amount

•Action 3: Output: Display the number of coins needed

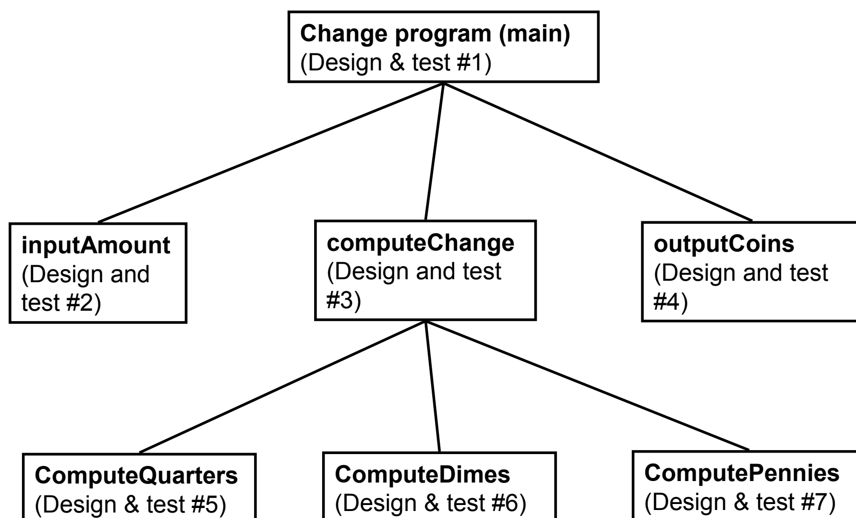
Program Design: An Example Problem

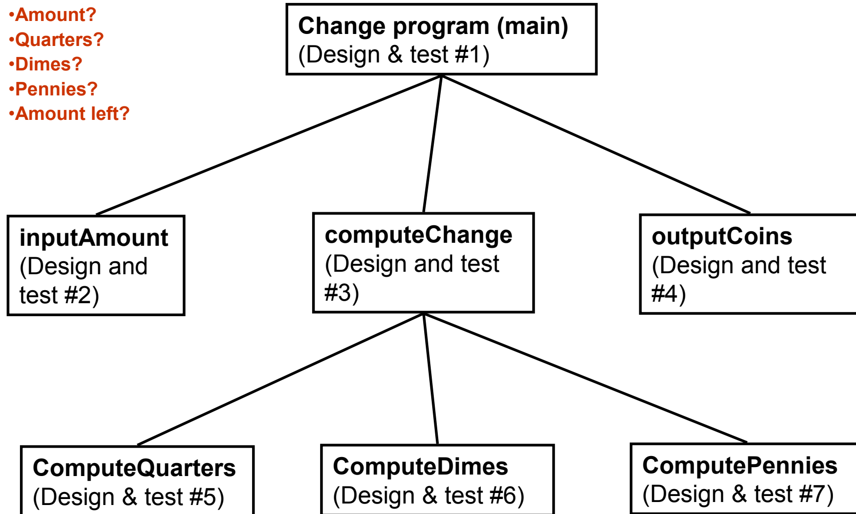
- However Action 2 (computing change) is still quite large and may require further decomposition into sub-actions.
- One sensible decomposition is:
  - Sub-action 2A: Compute the number of quarters to be given out.
  - Sub-action 2B: Compute the number of dimes to be given out.
  - Sub-action 2C: Compute the number of pennies to be given out.
- Rules of thumb for designing functions:
  1. Each function should have one well defined task. If it doesn't then it may have to be decomposed into multiple sub-functions.
     a) Clear function: A function that prompts the user to enter the amount of money.
     b) Ambiguous function: A function that prompts for the amount of money and computes the number of quarters to be given as change.
  2. Try to avoid writing functions that are longer than one screen in size (again this is a rule of thumb!)

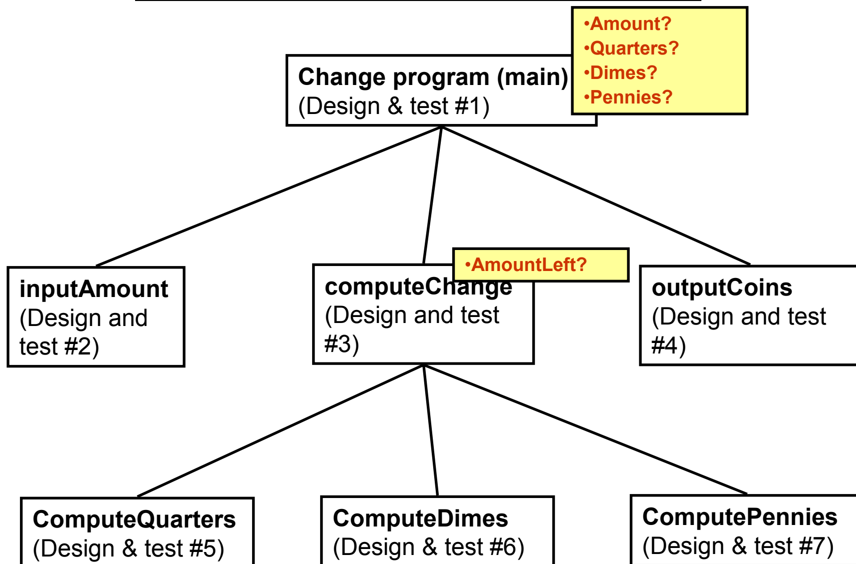
Determining What Information Needs To Be Tracked

1. Amount of change to be returned
2. Number of quarters to be given as change
3. Number of dimes to be given as change
4. Number dimes to be given as change
5. The remaining amount of change still left (changes as quarters, dimes and pennies are given out)

Outline Of The Program




Where To Declare Your Variables?
•Amount?
•Quarters?
•Dimes?
•Pennies?
•Amount left?
Change program (main)
(Design & test #1)
inputAmount
(Design and test #2)
computeChange
(Design and test #3)
outputCoins
(Design and test #4)
ComputeQuarters
(Design & test #5)
ComputeDimes
(Design & test #6)
ComputePennies
(Design & test #7)
James Tam


Where To Declare Your Variables?
•Amount?
•Quarters?
•Dimes?
•Pennies?
Change program (main)
(Design & test #1)
•AmountLeft?
inputAmount
(Design and test #2)
computeChange
(Design and test #3)
outputCoins
(Design and test #4)
ComputeQuarters
(Design & test #5)
ComputeDimes
(Design & test #6)
ComputePennies
(Design & test #7)
James Tam

Skeleton Functions

It's a outline of a function with a bare minimum amount that is needed to translate to machine (keywords required, function name, a statement to define the body – return values and parameters may or may not be included in the skeleton).

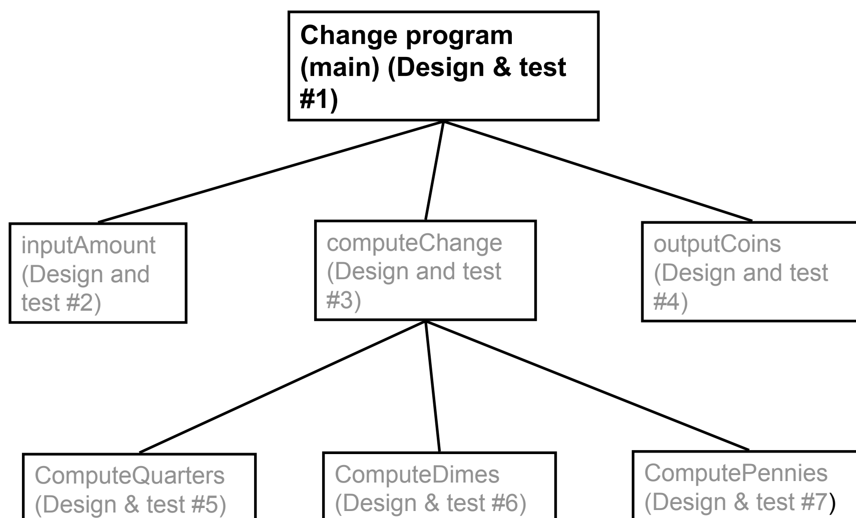
Code Skeleton: Change Maker Program

```
def inputAmount (amount):
    return amount

def computeQuarters (amount, amountLeft, quarters):
    return amountLeft, quarters

def computeDimes (amountLeft, dimes):
    return amountLeft, dimes

def computePennies (amountLeft, pennies):
    return pennies

def computeChange (amount, quarters, dimes, pennies):
    amountLeft = 0
    return quarters, dimes, pennies

def outputCoins (amount, quarters, dimes, pennies):
    print ""
```

Code Skeleton: Change Maker Program (2)

```
# MAIN FUNCTION
amount = 0
quarters = 0
dimes = 0
pennies = 0
```


Implementing And Testing The Main Function

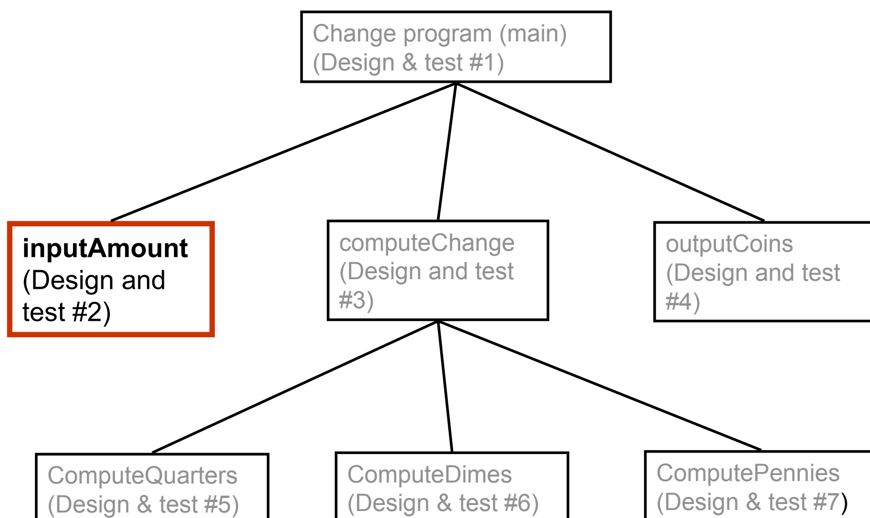


Implementing And Testing The Main Function

```
# MAIN FUNCTION
amount = 0
quarters = 0
dimes = 0
pennies = 0

write " Before 'inputAmount' "
amount = inputAmount (amount)
write " After 'inputAmount' "

# JT's note: Do the same tests to check the calls the other functions
quarters, dimes, pennies = computeChange (amount, quarters, dimes, pennies)
outputCoins (amount, quarters, dimes, pennies)
```

```
def inputAmount (amount):
   print "<<<Inside inputAmount>>>"
   return amount
```


Implementing And Testing InputAmount



Implementing And Testing Input Functions

 # Function definition

```
def inputAmount (amount):
    amount = input ("Enter an amount of change from 1 to 99 cents: ")
    return amount
```

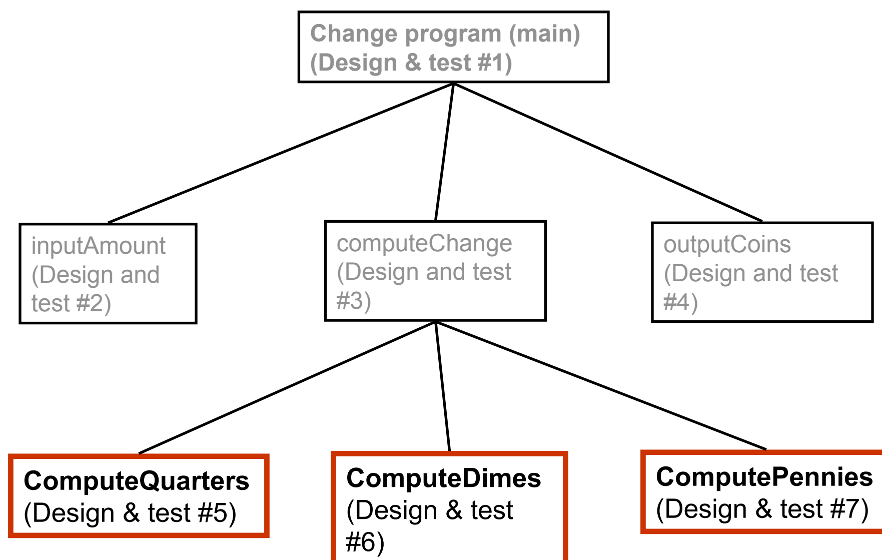
 # Testing the function definition

```
amount = inputAmount (amount)
print "amount:", amount
```

Test that your inputs were read in correctly DON'T ASSUME that they were!

James Tam

Implementing And Testing The Compute Functions



James Tam

Implementing And Testing ComputeQuarters

```
# Function definition
def computeQuarters (amount, amountLeft, quarters):
    quarters = amount / 25
    amountLeft = amount % 25
    return amountLeft, quarters


# Function test
amount = 0;
amountLeft = 0
quarters = 0
amount = input ("Enter amount: ")
amountLeft, quarters = computeQuarters (amount, amountLeft, quarters)
print "Amount:", amount
print "Amount left:", amountLeft
print "Quarters:", quarters
```

Check the program calculations against some hand calculations.

Globals

By default variables and constants which aren't declared as local are global.

```
num1 = 1

def fun ():
    num2 = 2

# MAIN FUNCTION
num3 = 3
CONSTANT_VALUE = 4
```

Local variable (declared within the body of a function).

Global variables

Global constant

Globals

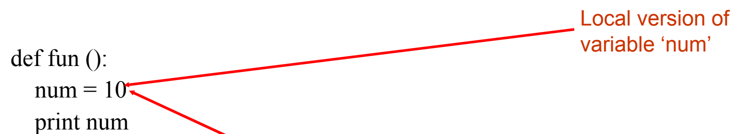
By default variables and constants which aren't declared as local are global.

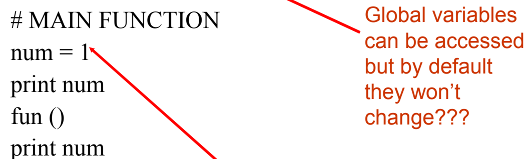
```
num1 = 1

def fun ():
    num2 = 2

    write num1
    write num3
    write CONSTANT_VALUE

# MAIN FUNCTION
num3 = 3
CONSTANT_VALUE = 4
```

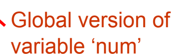
Avoid The Use Of Global Variables Without A Compelling Reason

1. Using global variables (without at least passing them as parameters) may result in programs that are harder to trace:

```
def fun ():
    num = 10
    print num

# MAIN FUNCTION
num = 1
print num
fun ()
print num
```







Avoid The Use Of Global Variables Without A Compelling Reason

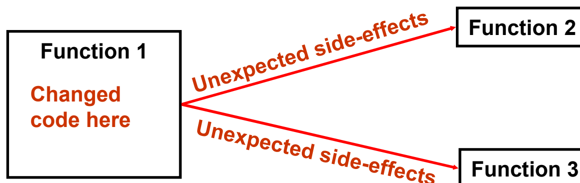
1. Using global variables (without at least passing them as parameters) may result in programs that are harder to trace:

```
def fun ():
    global num
    num = 10
    print num

# MAIN FUNCTION
num = 1
print num
fun ()
print num
```

An explicit indicator that all references to 'num' are to the global variable not a local version.

Avoid The Use Of Global Variables Without A Compelling Reason

2. Also programs that make extensive use of global variables may be harder to maintain and update:
   - Changes in one part of the program may produce unexpected side effects in other parts of the program that weren't changed with the update.



Globals: Bottom Line

Using global constants is acceptable and is often regarded as good programming style.

```
def calculateChange (population):
    population = population + (population * (BIRTH_RATE - MORTALITY_RATE))
    return population

def displayChange (population):
    print "Birth rate:", BIRTH_RATE, ", mortality rate:", MORTALITY_RATE
    print "Projected population:", population

# MAIN FUNCTION
BIRTH_RATE = 0.1
MORTALITY_RATE = 0.01
population = 100

print "Existing population:", population
population = calculateChange (population)
displayChange (population)
```


Globals: Bottom Line (2)

Using global constants is acceptable and is often regarded as good programming style.

```
def calculateChange (population):
    population = population + (population * (BIRTH_RATE - MORTALITY_RATE))
    return population

def displayChange (population):
    print "Birth rate:", BIRTH_RATE, ", mortality rate:", MORTALITY_RATE
    print "Projected population:", population

# MAIN FUNCTION
BIRTH_RATE = 0.5
MORTALITY_RATE = 0.001
population = 100

print "Existing population:", population
population = calculateChange (population)
displayChange (population)
```

Globals: Bottom Line (3)

However the use of global variables should be largely avoided:

- Programs are harder to trace and read
- Maintenance may be harder

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
- Easier to test the program
- Easier to maintain (if functions are independent changes in one function can have a minimal impact on other functions)

You Now Know

- How to write the definition for a function
  - How to write a function call
- How to pass information to and from functions via parameters and return values
- What is the difference between a local variable/constant and a global
- How to test functions and procedures
- How to design a program from a problem statement
  - How to determine what are the candidate functions
  - How to determine what variables are needed and where they need to be declared
  - Some approaches for developing simple algorithms