CPSC 231: Functions: Decomposition And Code Reuse

You will learn 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/2018/231W/examples/decomposition
- Location (via the CPSC UNIX network):
  - /home/231/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 sequential order.
- Instead execution will appear to 'jump around' so it will be harder to follow the examples if you don't do a little preparatory work.

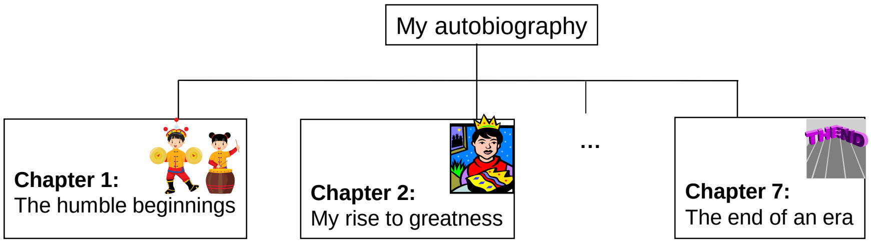
- Also it would be helpful to take notes that include greater detail:
  - For example: Literally just sketching out the diagrams that I draw without the extra accompanying verbal description that I provide in class probably won't be useful to study from later.

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)



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

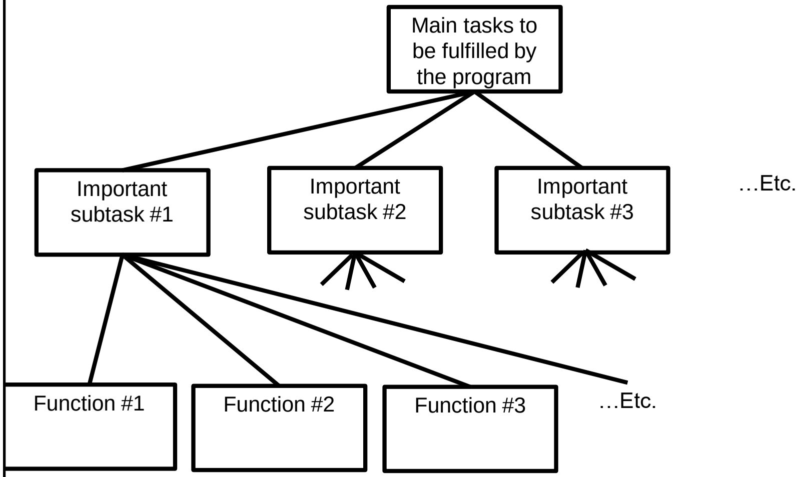
It all started ten and one score years ago with a log-shaped computer work station…

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



Main tasks to be fulfilled by the program

Important subtask #1

Important subtask #2

Important subtask #3

…Etc.

Function #1

Function #2

Function #3

…Etc.

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.

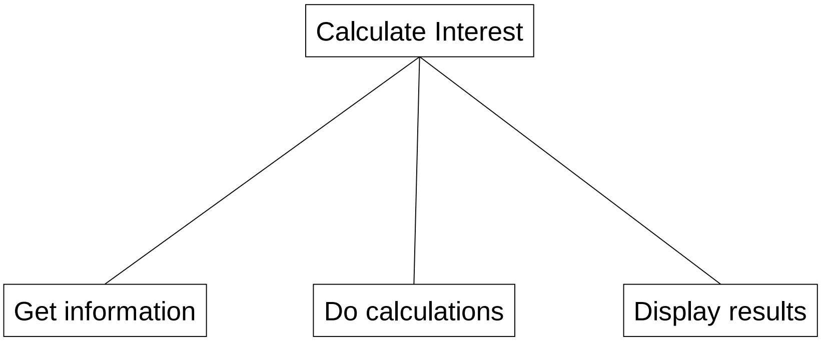
Example Problem

- 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)



Calculate Interest

Get information

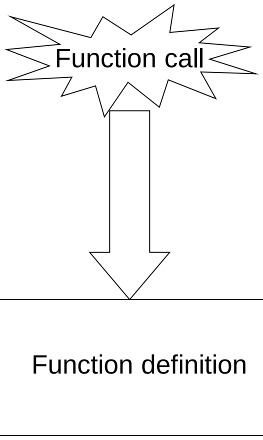
Do calculations

Display results

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)



Function call

Function definition

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 (when called).

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

Calling A Function

- **Format:**

  <function name>()

- **Example**:

  ```
  displayInstructions()
  ```

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 statement have been un-indented (save loops/branches) so it’s just the first statement that is the starting execution point.

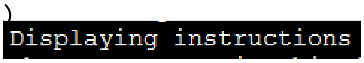
```
HUMAN_CAT_AGE_RATIO = 7
age = input("What is your age in years: ")
catAge = age * HUMAN_CAT_AGE_RATIO
…
```

- 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

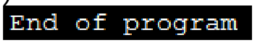
- Name of the example program: `1firstExampleFunction.py`

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



```
Displaying instructions
```

```
# Main body of code (starting execution point, not indented)
displayInstructions()
print("End of program")
```



```
End of program
```

James Tam

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

James Tam

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) “**`2firstExampleFunctionV2.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! Program starts at the first executable un-indented instruction

James Tam

Stylistic Note

- By convention the starting function is frequently named ‘`main()`’ or in my case ‘`start()`’.

```
def main():
```

- OR

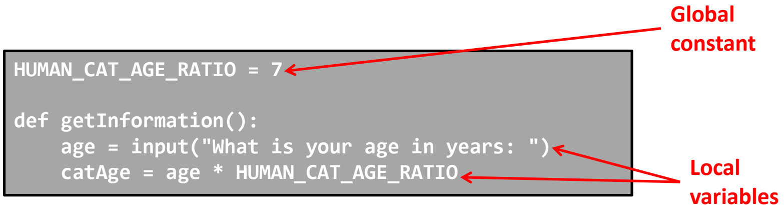
```
def start():
```

- This is done so the reader can quickly find the beginning execution point.

James Tam

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

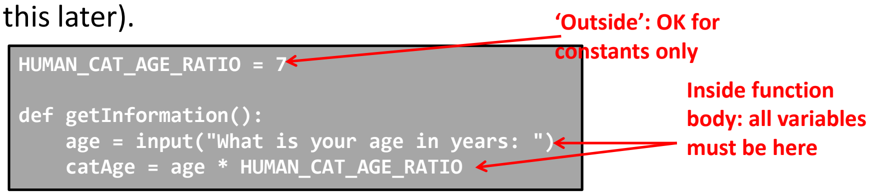
Local variables

Creating Your Variables

- Before this section of notes: 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
```

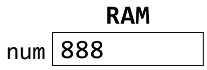
'Outside': OK for constants only

Inside function body: all variables must be here

What You Know: Declaring Variables

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

```
num = 888
```



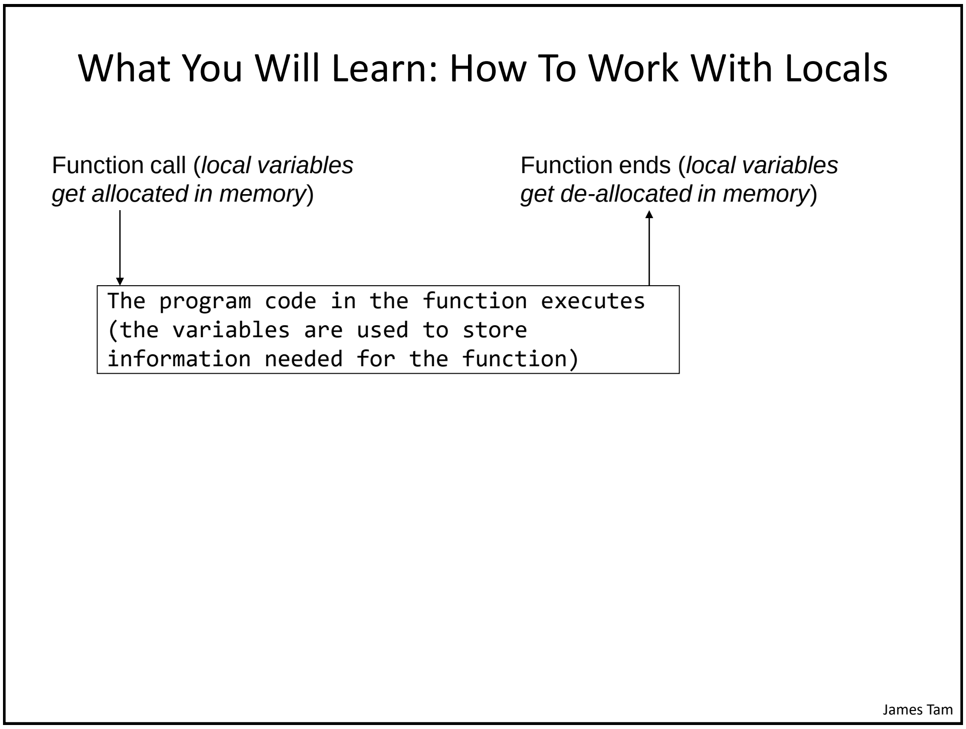
RAM

num 888

- 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: What Is The Significance Of Being ‘Local’

- To minimize the amount of memory that is used to store the contents of variables only create variables when they are needed (“allocated”).
- When the memory for a variable is no longer needed it can be ‘freed up’ and reused (“de-allocated”).
- 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.
- (There’s an even better reason for making variables local coming up later ‘side effects’)



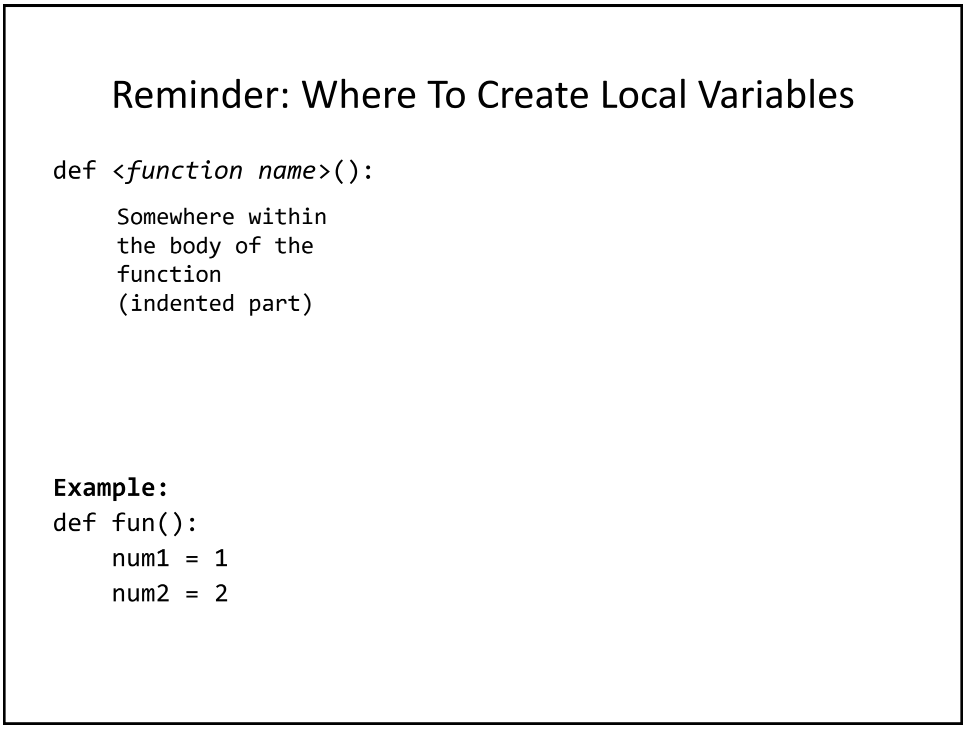
What You Will Learn: How To Work With Locals

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)
```

James Tam



Reminder: Where To Create Local Variables

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

Example:

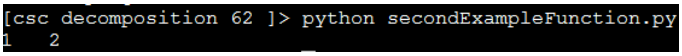
```
def fun():
    num1 = 1
    num2 = 2
```

Working With Local Variables: Putting It All Together

- Name of the example program: **3secondExampleFunction.py**

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

# start function
fun()
```

Variables that are local to function 'fun'

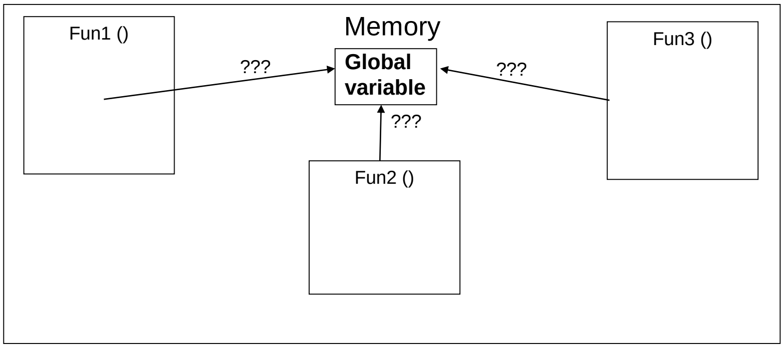


```
[csc decomposition 62 ]> python secondExampleFunction.py
1    2
```

James Tam

Another Reason For Creating Local Variables

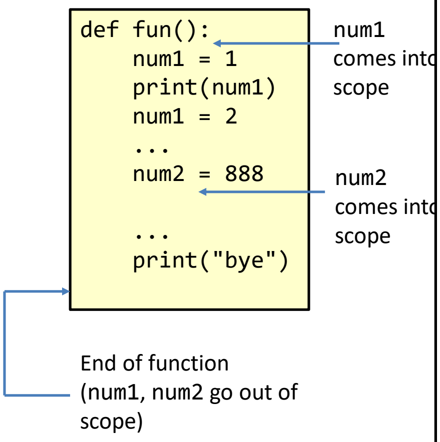
- To minimize side effects (unexpected changes that have occurred to variables after a function has ended e.g., a variable storing the age of the user accidentally takes on a negative value).
- To visualize the potential problem: imagine if all variables could be accessed anywhere in the program (not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



James Tam

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.



```
def fun():
    num1 = 1
    print(num1)
    num1 = 2
    ...
    num2 = 888

    ...
    print("bye")
```

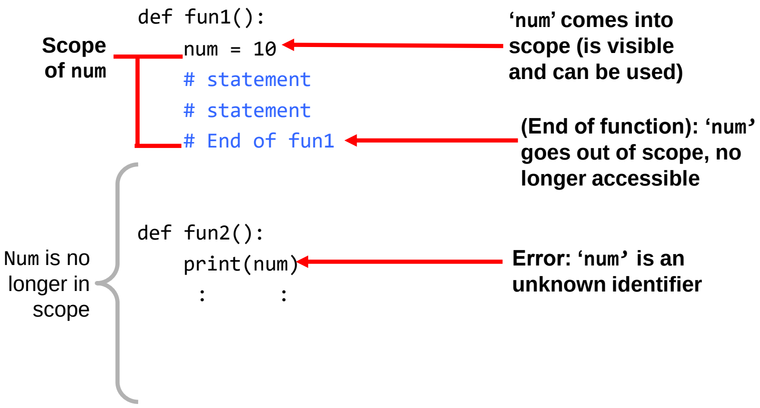
num1 comes int scope

num2 comes int scope

End of function (num1, num2 go out of scope)

[1] The concept of scoping (limited visibility) applies to all programming languages. The rules for determining when identifiers come into and go out of scope will vary with a particular language.

Identifiers Out Of Scope: An Example



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

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

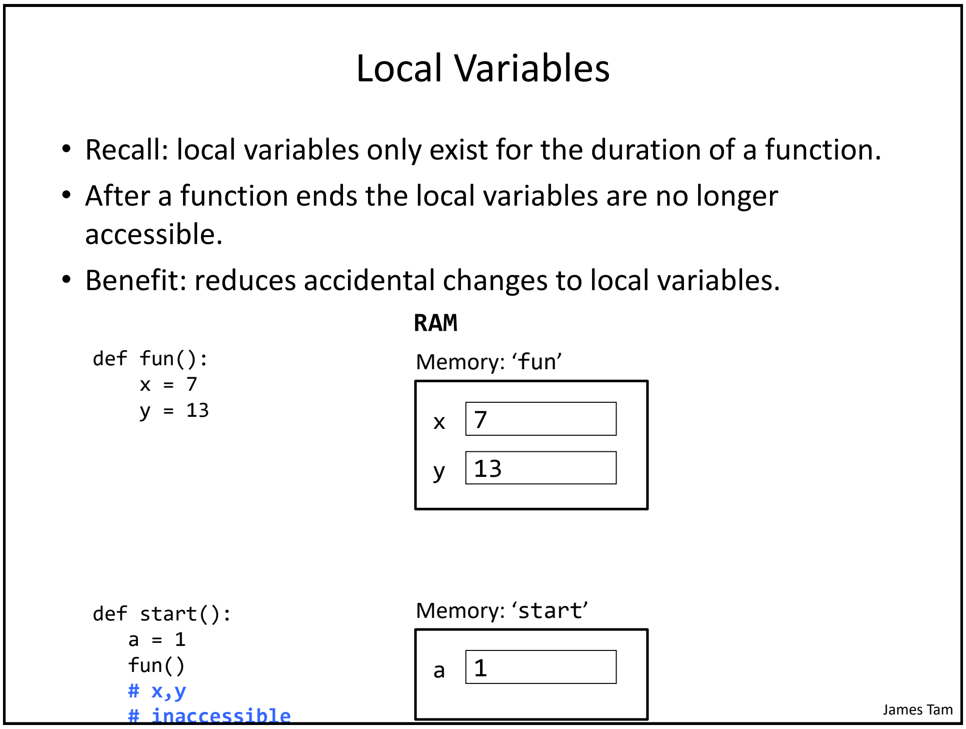
Scope of num

'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

Error: 'num' is an unknown identifier



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.

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

RAM

Memory: 'fun'

x 7

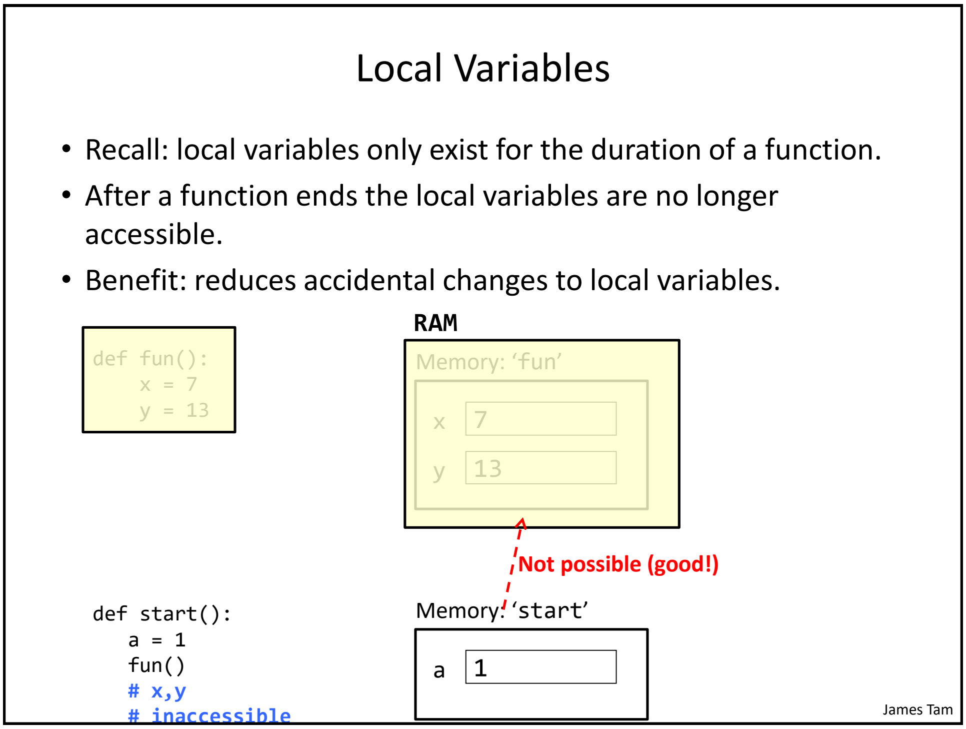
y 13

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

Memory: 'start'

a 1

James Tam



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.

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

RAM

Memory: 'fun'

x 7

y 13

Not possible (good!)

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

Memory: 'start'

a 1

James Tam

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’???

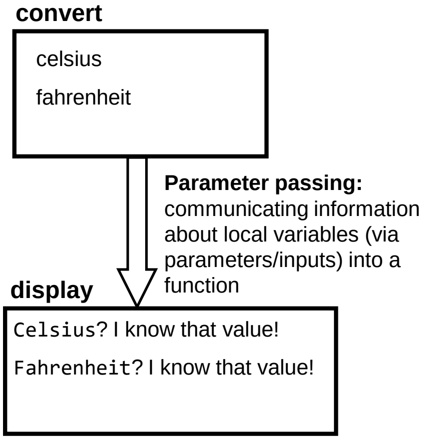
Variables celsius and fahrenheit are local to function ‘convert()’

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

James Tam

One Solution: Parameter Passing

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



convert

celsius

fahrenheit

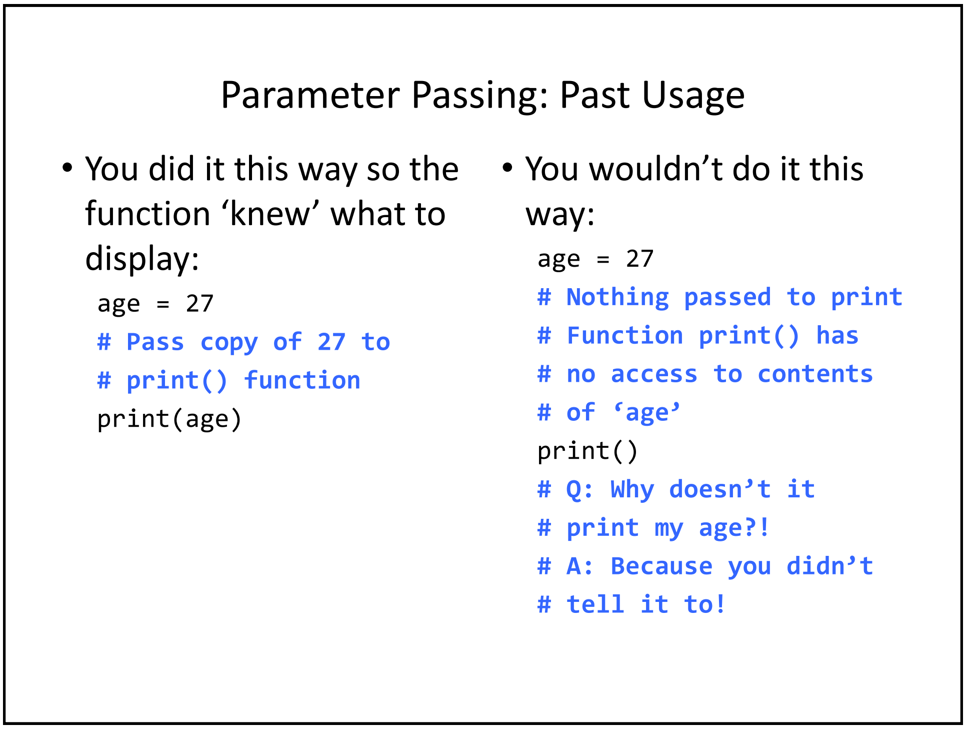
Parameter passing: communicating information about local variables (via parameters/inputs) into a function

display

Celsius? I know that value!

Fahrenheit? I know that value!

James Tam



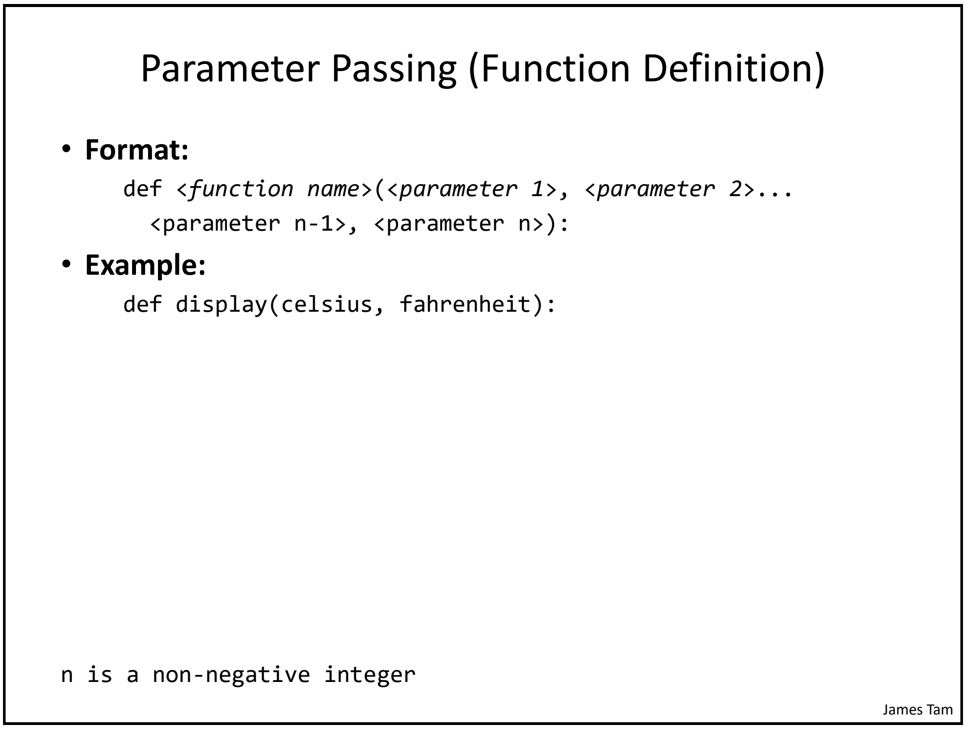
Parameter Passing: Past Usage

- 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 (Function Definition)

- **Format:**

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

- **Example:**

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

n is a non-negative integer

James Tam

Parameter Passing (Function Call)

- **Format:**

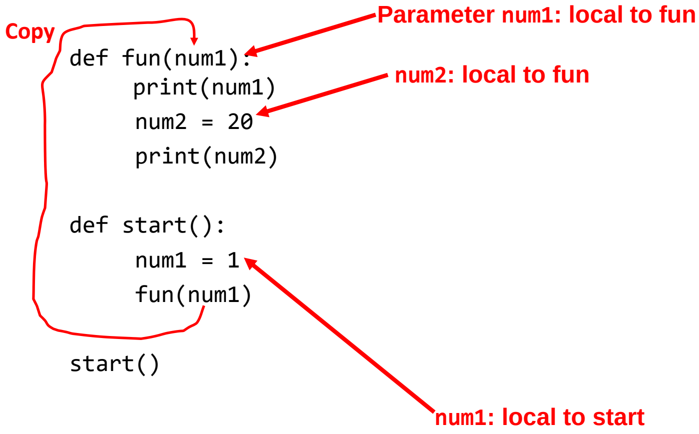
```
<function name>(<parameter 1>, <parameter 2>...
    <parameter n-1>, <parameter n>)
```

- **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)

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

start()
```

Important Terminology

- Getting **user input**:
  - The user “types in” the information
  - In Python the **`input()`** function is employed
- Passing **inputs/parameters** into a function
  - Information passed into a function as the function runs

Format:

```
                Inputs/parameters
                      ↓
<Function name>(           )
```

Examples:

```
print("hello")    # input = "hello"
random.randrange(6)  # input = 6
print()   # No input
round(3.14,1)  # 2 inputs = 3.14(data), 1(# fraction digits)
```

James Tam

Sample (Simple) Example Question: Terminology

- Write a function that takes two inputs: a numerator and denominator
- The function will calculate and display onscreen the floating point quotient
- Solution:
  - There is no mention of *user* input
    - Don’t call the `input()` function!
  - Consequently the input in the program description refers to information passed into the function as it runs

```
# Correct function definition
def aFunction(numerator,denominator):
    quotient = numerator/denominator
    print(quotient)
```

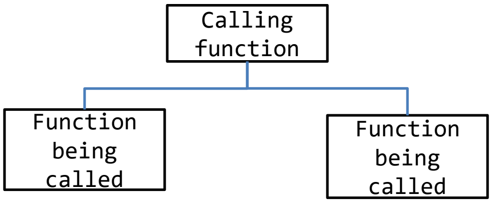
James Tam

In Class Exercise:

- Write a function that takes as input two parameters.
- The function will display the two parameters using the `print()` function and display them on one line separated by a space.

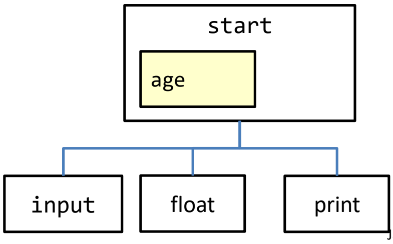
Structure Charts

- Useful for visualizing the layout of function calls in a large and complex program.
- **Format**:



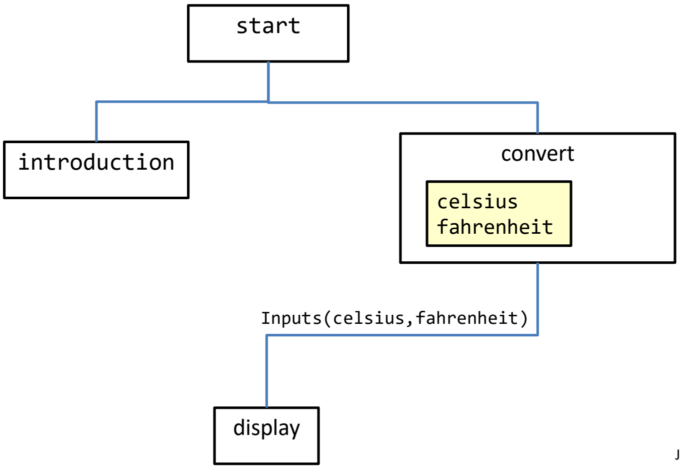
- **Example**:

```
def start():
    age = float(input())
    print(age)
```



Structure Chart: `temperature.py`

- To reduce clutter most structure charts only show functions that were directly implemented by the programmer (or team).



start

introduction

convert

celsius
fahrenheit

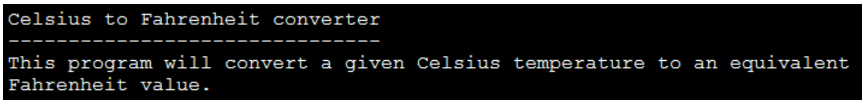
Inputs(celsius,fahrenheit)

display

Parameter Passing: Putting It All Together

- Name of the example program: **`4temperature.py`**

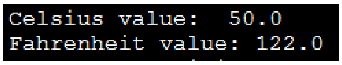
```
def introduction():
    print("""
Celsius to Fahrenheit converter
-------------------------------
This program will convert a given Celsius temperature to an
  equivalent
Fahrenheit value.

    """)
```



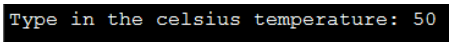
```
Celsius to Fahrenheit converter
-------------------------------
This program will convert a given Celsius temperature to an equivalent
Fahrenheit value.
```

Parameter Passing: Putting It All Together (2)

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



```
Celsius value:   50.0
Fahrenheit value: 122.0
```



```
Type in the celsius temperature: 50
```

```
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()
```

James Tam

Parameter Passing: Important Recap!

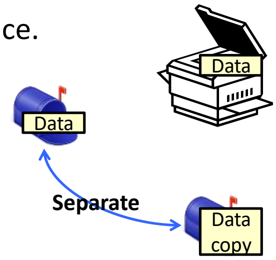
- A parameter is copied into a local memory space.

```
# Inside function convert()
display(celsius, fahrenheit)      # Function call
```

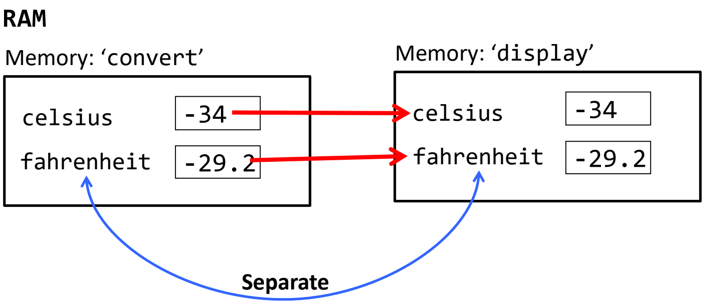
Make copy

Make copy

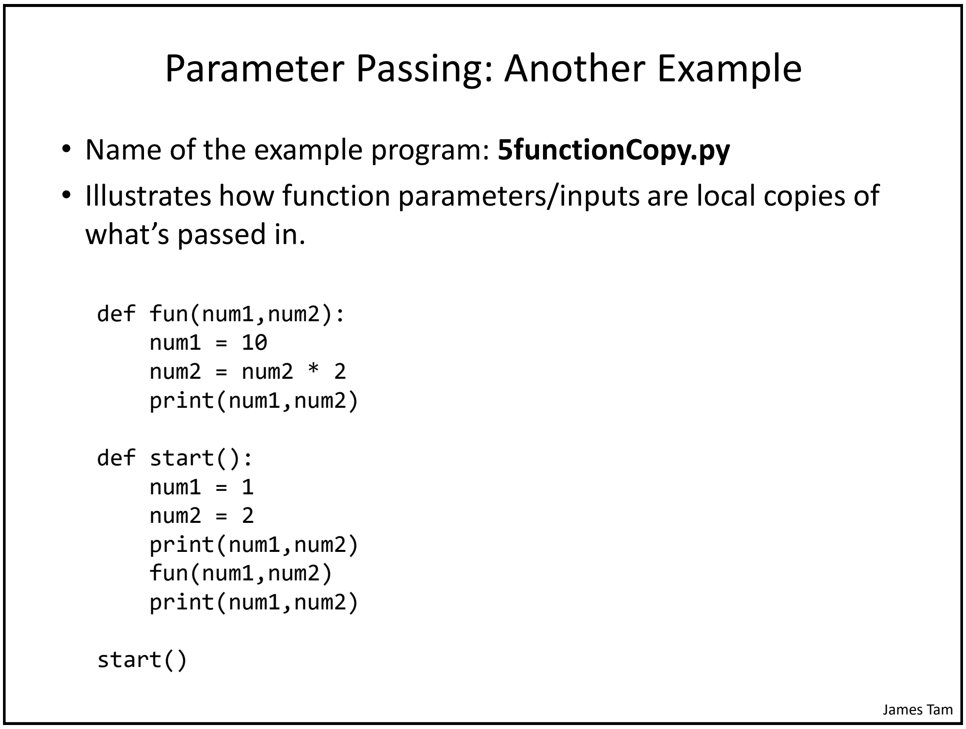
```
# Inside function display
def display(celsius, fahrenheit):   # Function
                                    # definition
```



RAM



James Tam

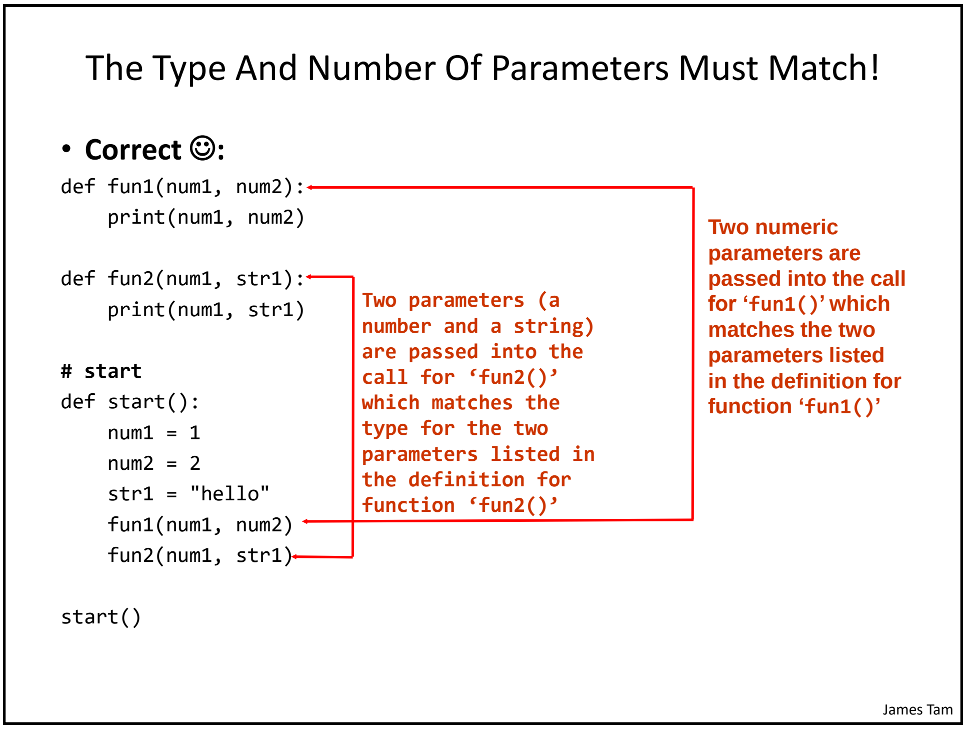


Parameter Passing: Another Example

- Name of the example program: **5functionCopy.py**
- Illustrates how function parameters/inputs are local copies of what’s passed in.

```
def fun(num1,num2):
    num1 = 10
    num2 = num2 * 2
    print(num1,num2)

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

start()
```

James Tam



The Type And Number Of Parameters Must Match!

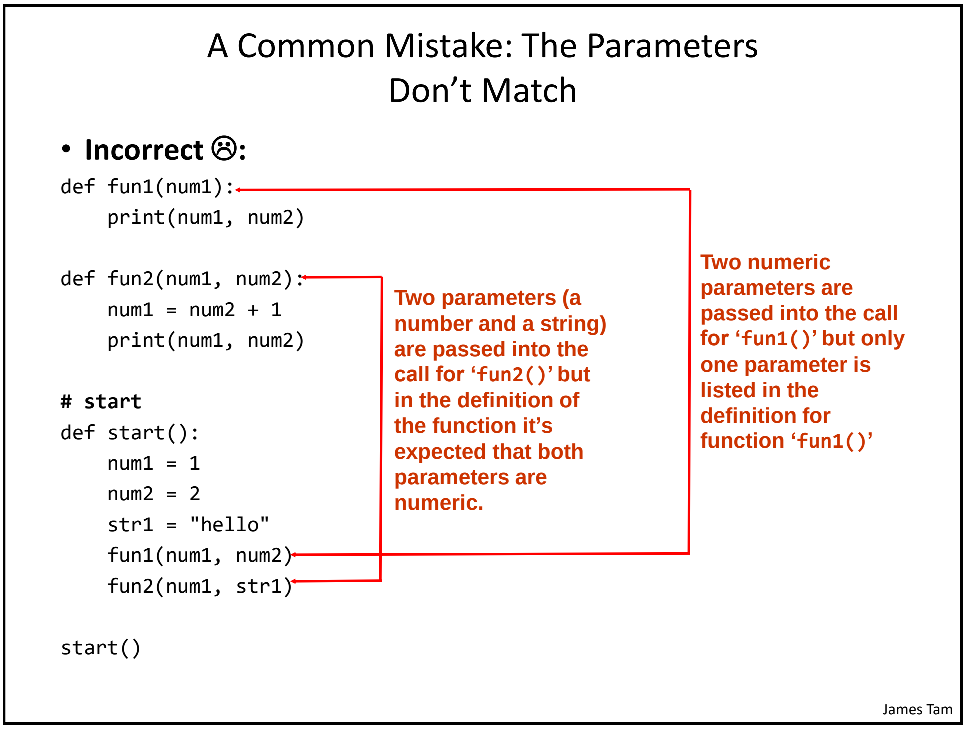
- **Correct ☺:**

```
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()’

James Tam



A Common Mistake: The Parameters Don't Match

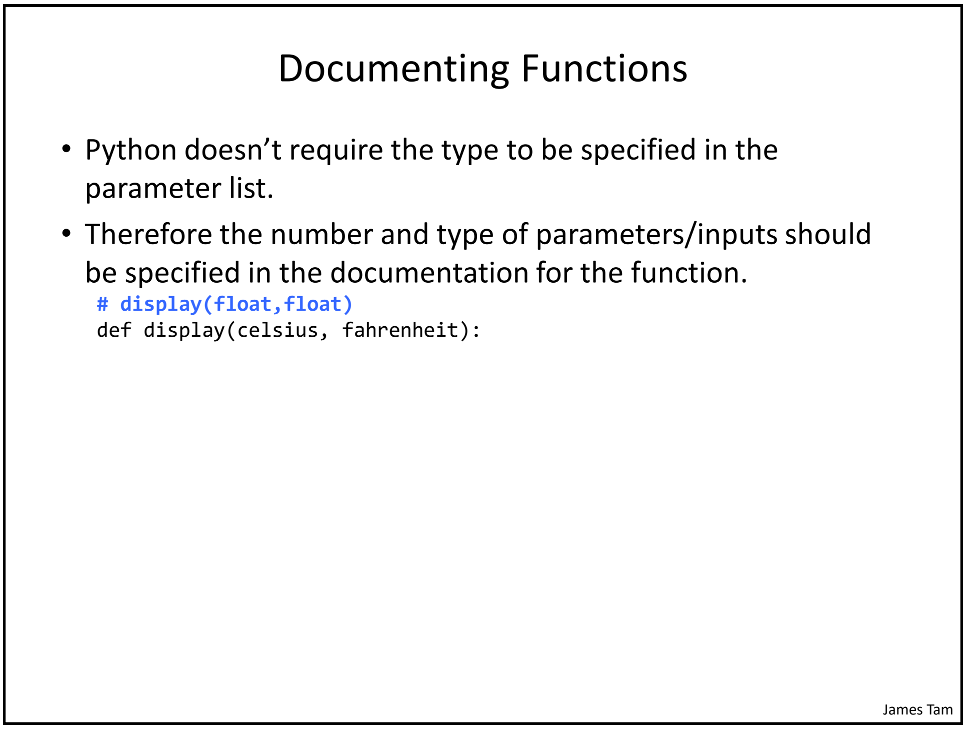
- **Incorrect ☹:**

```
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()`'

James Tam



Documenting Functions

- Python doesn't require the type to be specified in the parameter list.
- Therefore the number and type of parameters/inputs should be specified in the documentation for the function.

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

James Tam

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()
```

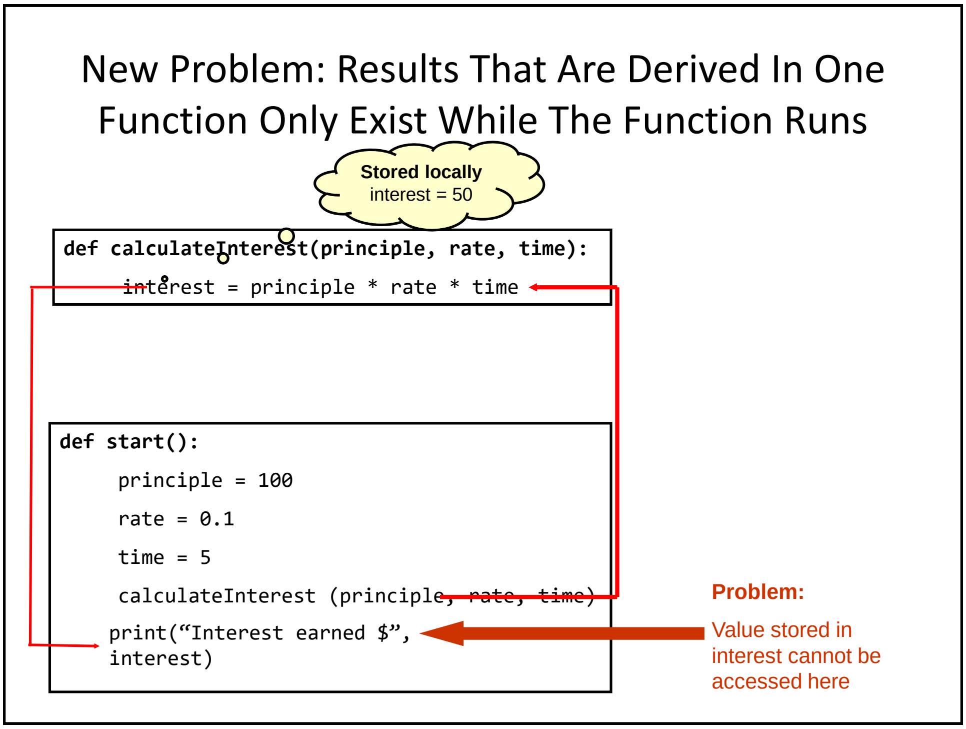
What is 'num'? It has not been created in function 'start()'

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

start()
```

Scope: A Variant Example

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

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

- **What happens at this point?**
- **Why?**



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

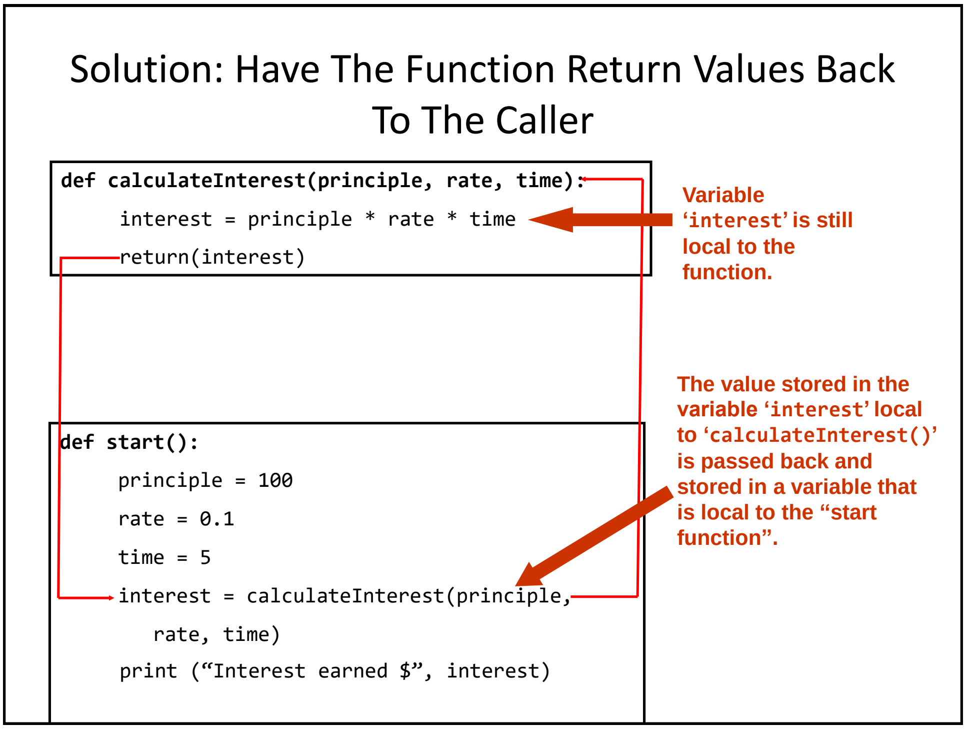
Stored locally
interest = 50

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

```
def 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.

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

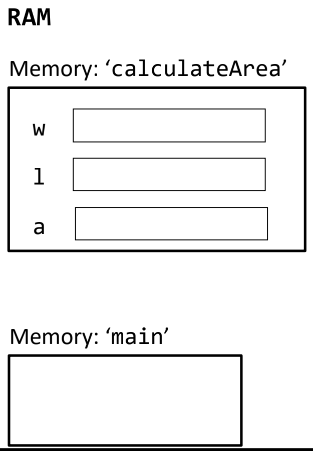
The value stored in the variable 'interest' local to 'calculateInterest()' is passed back and stored in a variable that is local to the "start function".

Function Return Values (1)

- Remember that local variables only exist for the duration of a function.

```
def calculateArea():
    w = int(input())
    l = int(input())
    a = w * l
```

```
def main():
   calculateArea()
   print(area)
```

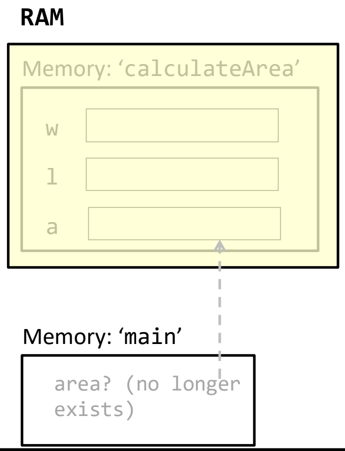


Function Return Values (2)

- After a function has ended local variables are 'gone'.

```
def calculateArea():
    w = int(input())
    l = int(input())
    a = w * l
```

```
def main():
   calculateArea()
   print(area)
```

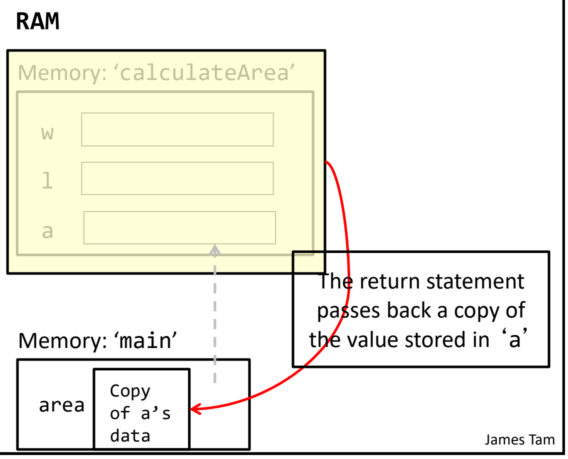


Function Return Values (3)

- Function return values communicate a copy of information out of a function (back to the caller) just as the function ends.

```
def calculateArea():
    w = int(input())
    l = int(input())
    a = w * l
    return(a)
```

```
def main():
   area = calculateArea()
   print(area)
```



The return statement passes back a copy of the value stored in 'a'

James Tam

Using Return Values

- **Format (Single value returned)[1]:**

```
return(<value returned>)              # Function definition
<variable name> = <function name>()   # Function call
```

- **Example (Single value returned) [1]:**

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

[1] Although bracketing the return value isn't required when only a single value is returned it's still recommended that you get in the habit of doing it because it is required for 'multiple' return values. The actual details about the difference between returning a single vs. 'multiple' values will be covered in the 'composites' section.

Using Return Values

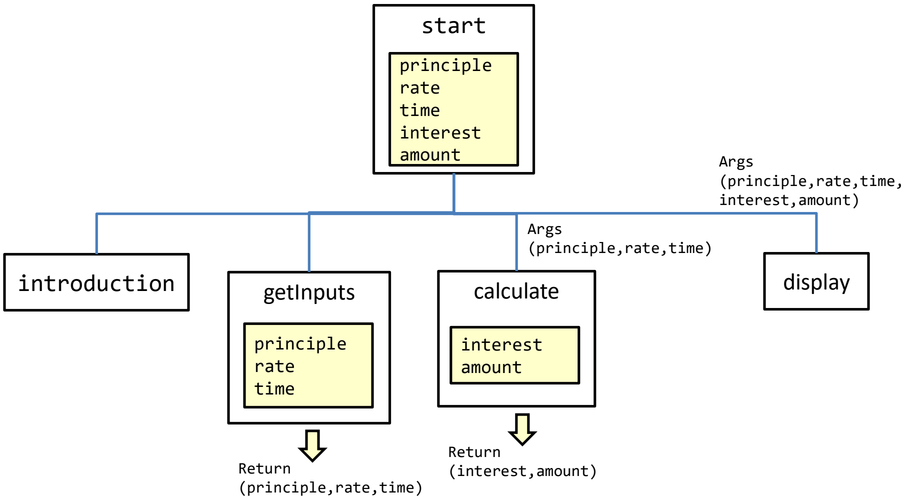
- **Format (Multiple values returned):**

```
# 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()
```

Structure Chart: interest.py



start

principle
rate
time
interest
amount

Args
(principle,rate,time,
interest,amount)

Args
(principle,rate,time)

introduction

getInputs

principle
rate
time

Return
(principle,rate,time)

calculate

interest
amount

Return
(interest,amount)

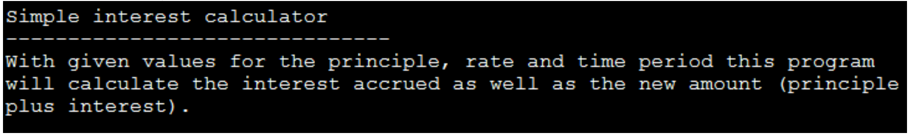
display

James Tam

Using Return Values: Putting It All Together

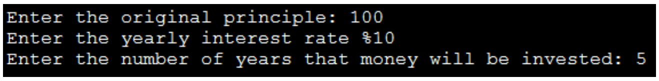
- Name of the example program: **6interest.py**

```
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).
    """)
```



```
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)

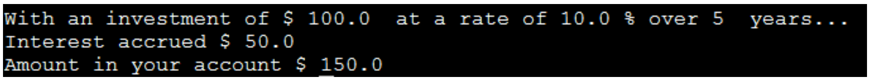


```
Enter the original principle: 100
Enter the yearly interest rate %10
Enter the number of years that money will be invested: 5
```

```
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)
```



```
With an investment of $ 100.0  at a rate of 10.0 % over 5  years...
Interest accrued $ 50.0
Amount in your account $ 150.0
```

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()
```

Stylistic Note

- Creating variables all at once at the start of a 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()
```

Not syntactically required but a stylistic approach

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)
```

Documenting Functions

- Similar to specifying the function parameters/inputs, the type of the return values should also be documented.
- Example:

```
# calculate
# returns(float,float)
def calculate(principle, rate, time):
```

James Tam

Another Common Mistake: Not Saving Return Values (Pre-Created Functions)

- You would typically never use the `input()` function this way
- (Function return value not stored)

```
input("Enter your name")
print(name)
```

- (Function return value should be stored)

```
name = input("Enter your name")
print(name)
```

James Tam

Yet Another Common Mistake: Not Saving Return Values (Your Functions)

- Just because a function returns a value does not automatically mean the return 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

- Function return values must be explicitly saved by the caller of the function.

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

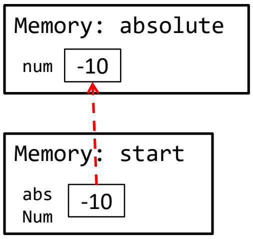
```
# Start: error
area = 0
calculateArea(4,3)
print(area)
```

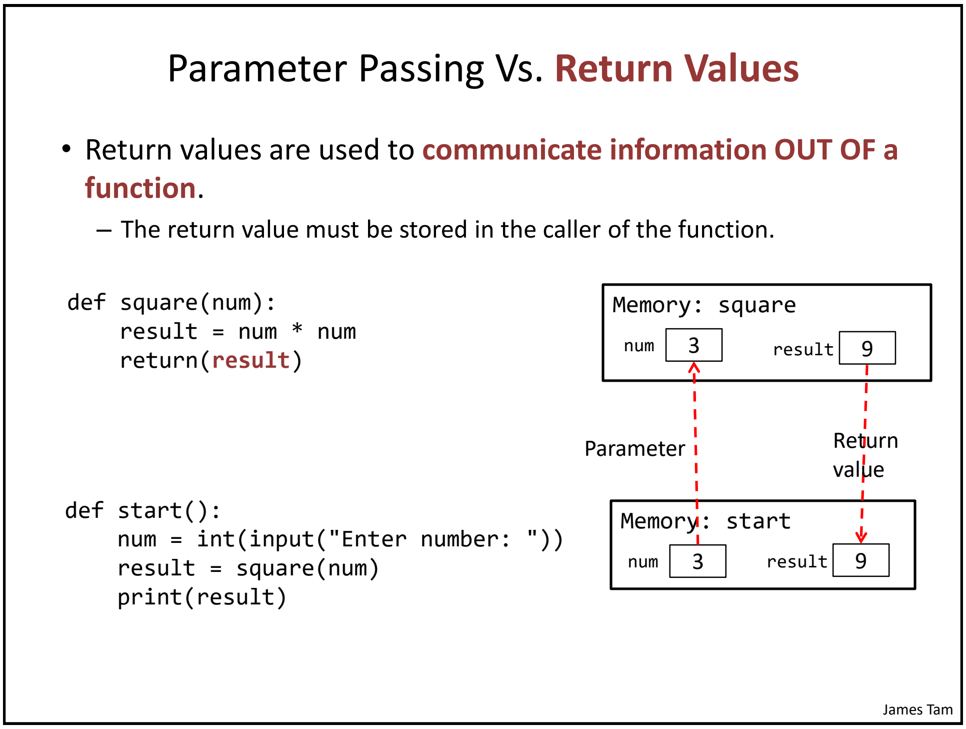
```
# Start: fixed
area = 0
area = calculateArea (4,3)
print(area)
```

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: "))
    absNum = (absolute(num))
    print(absNum)
```





Parameter Passing Vs. **Return Values**

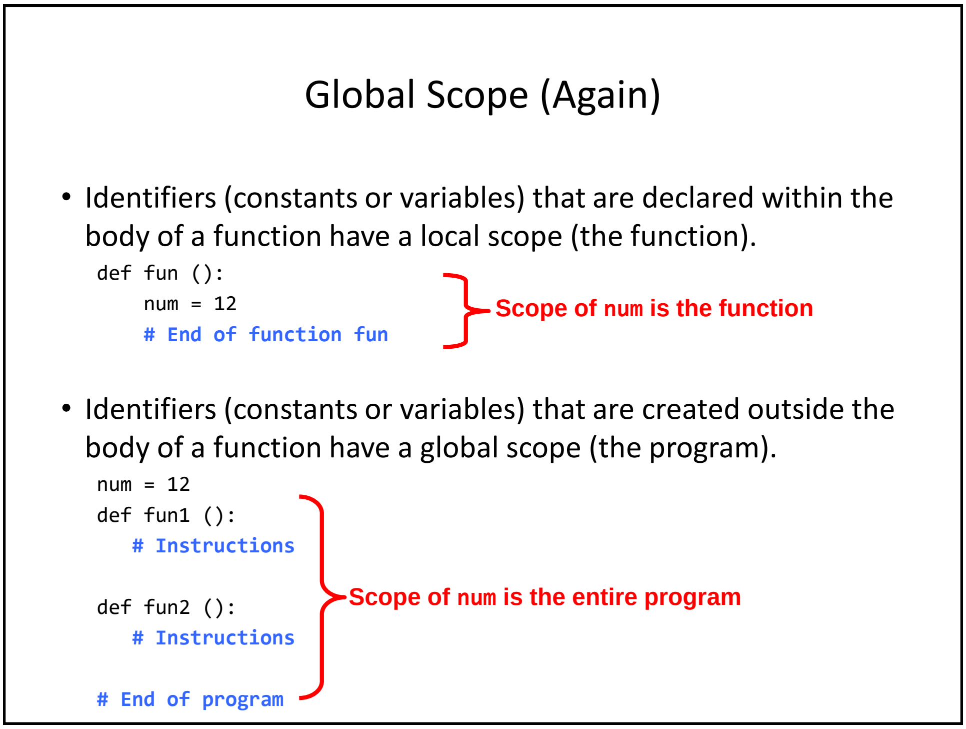
- Return values are used to **communicate information OUT OF a function**.
  - The return value must be stored in the caller of the function.

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

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

Memory: square
num 3
result 9
Parameter
Return value
Memory: start
num 3
result 9

James Tam



Global Scope (Again)

- 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 created outside the body of a function have a global scope (the program).

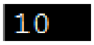
```
num = 12
def fun1 ():
   # Instructions

def fun2 ():
   # Instructions

# End of program
```

Scope of num is the entire program

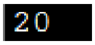
Global Scope: An Example

- Name of the example program: **7globalExample1.py**

```
num1 = 10

def fun():
   print(num1)

def start():
   fun()
   print(num2)

num2 = 20

start()
```



```
10
```



```
20
```

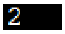
Global Variables: General Characteristics

- 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.
    - Unless you are told otherwise using global variables can affect the style component of your assignment grade.
    - Global constants are acceptable and are commonly used.

Global Variables: Python Specific Characteristic

- Name of the example program: **`8globalExample2.py`**

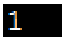
```
num = 1

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



```
2
```

Local created and displayed

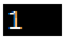
```
def start():
    print(num)
```



```
1
```

Global

```
    fun()
    print(num)
```



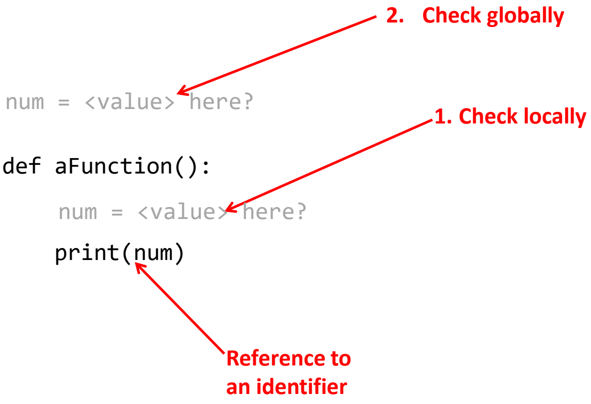
```
1
```

Global

```
start()
```

Scoping Rules: Globals

- When an identifier is referenced (variable or constant) then:
  1. First look in the local scope for the creation of the identifier: if found here then stop looking and use this identifier
  2. If nothing exists at the local level then look globally



```
num = <value> here?

def aFunction():
     num = <value> here?
    print(num)
```

2. Check globally

1. Check locally

Reference to an identifier

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 to the global, only the local is changed).

```
num = 1

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

Global num

Local num

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

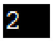
`global <`*`variable name`*`>`

Globals: Another Example (‘Write’ Access Via The “`Global`” Keyword)

- Name of the example program: **`9globalExample3.py`**

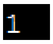
```
num = 1

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

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

start()
```

References to the name ‘num’ now affect the global variable, local variable not created



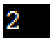
```
2
```

Global changed



```
1
```

Global

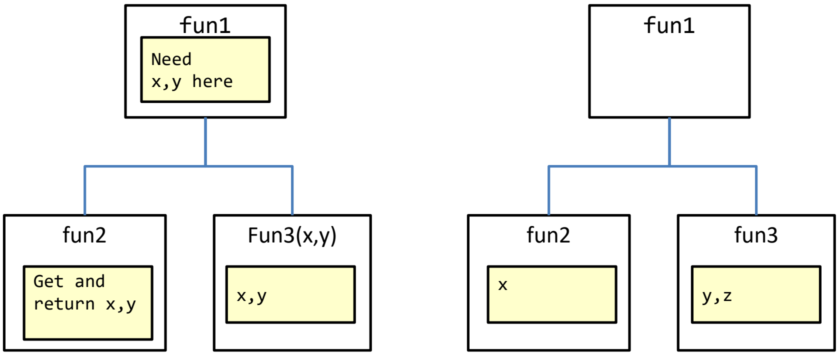


```
2
```

Global still changed after ‘`fun()`’ is done

What Level To Declare Variables

- Declare your variables as local to a function.
- When there are multiple levels of functions (a level is formed when one function calls another) then:
  - A variable should be created at the lowest level possible



James Tam

Documenting Functions

- (As previously mentioned the documentation should include)
  - The type and number of parameterss/inputs e.g., # fun(int,string)
  - The type and number of return values e.g., # returns(float,float,int)
- Additional documentation
  - 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 e.g., if a function will divide two parameters then the documentation should indicate that the function requires that the denominator is not zero.
    - (Authorship and version number may or may not be necessary for the purposes of this class although they are often included in actual practice).

James Tam

Doc Strings (If There Is Time)

- A special form of documentation:
  - **Characteristic 1**: It allows for documentation to span multiple lines
  - Example:

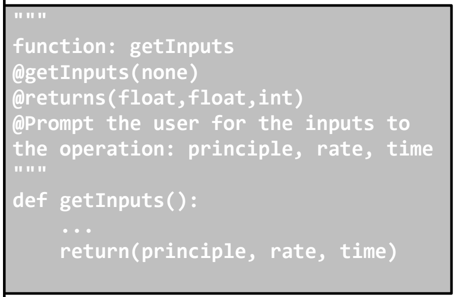
""" (triple double quotes)

```
function: getInputs
@getInputs(none)
@returns(float,float,int)
@Prompt the user for the inputs to the operation:
principle, rate, time
"""
def getInputs():
    ...
    return(principle, rate, time)
```

Doc Strings (If There Is Time, 2)

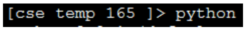
- **Characteristic 2**: it can provide help as the program is running in Python's interactive mode.
- Example: program is stored in file called "**doc_string.py**"
- Interactive mode is invoked by typing "python" at the command line (no program name)

doc_strings.py



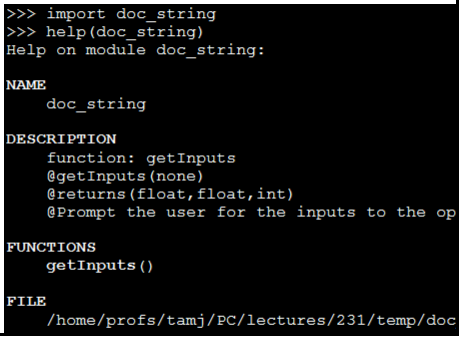
```
"""
function: getInputs
@getInputs(none)
@returns(float,float,int)
@Prompt the user for the inputs to
the operation: principle, rate, time
"""
def getInputs():
    ...
    return(principle, rate, time)
```

Start interactive mode



```
[cse temp 165 ]> python
```

Viewing help (doc string)



```
>>> import doc_string
>>> help(doc_string)
Help on module doc_string:

NAME
    doc_string

DESCRIPTION
    function: getInputs
    @getInputs(none)
    @returns(float,float,int)
    @Prompt the user for the inputs to the op

FUNCTIONS
    getInputs()

FILE
    /home/profs/tamj/PC/lectures/231/temp/doc
```

Boolean Functions

- Return a Boolean value (true/false): “Asks a question”
- Typically the Boolean function will ‘ask the question’ about a parameter(s)
- Example:
  - Is it true that the string can be converted to a number?

```
aString = input("Enter age: ")
ageOK = isNum(aString)
if (ageOK != True):
   print("Age must be a numeric value")
else:
   # OK to convert the string to a number
   age = int(aString)
```

```
# Boolean function
def isNum(aString):
   # Returns (True
   # or False)
```

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()`

James Tam

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()`

James Tam

Functions Should Be Defined Before They Can Be Called!

- **Correct** ☺

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

Function definition

```
# start
fun()
```

Function call

- **Incorrect** ☹

```
# Start
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()
```

The missing set of brackets do not produce a syntax/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 start ():
print ("start")

start ()
```

Another Common Problem: Indentation (2)

- Inconsistent indentation:

```
def start():
  print("first")
    # Error: Unless this is the body of branch or loop
    print("second")

start()
```

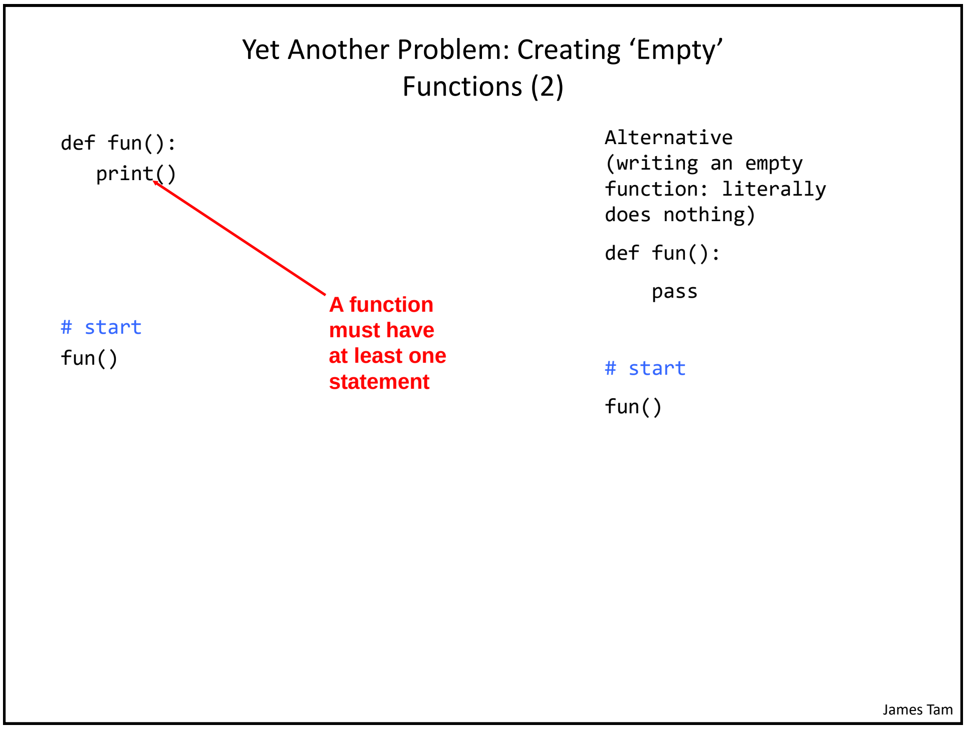
James Tam

Yet Another Problem: Creating ‘Empty’ Functions

```
def start():



start()
```

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

James Tam



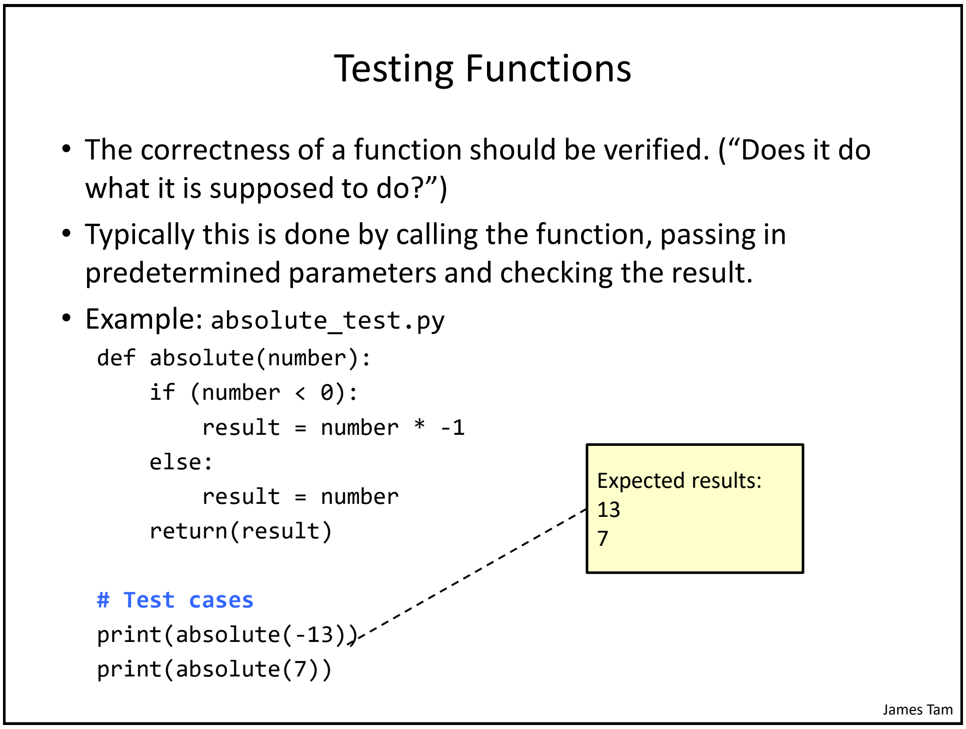
Yet Another Problem: Creating 'Empty' Functions (2)

```
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()
```

James Tam



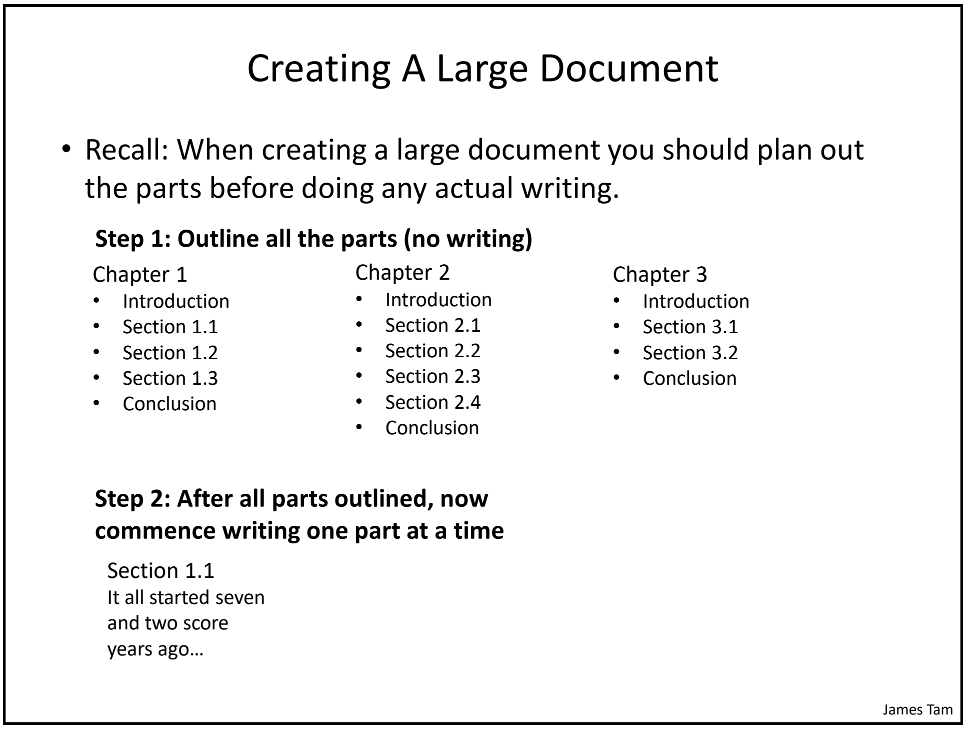
Testing Functions

- The correctness of a function should be verified. ("Does it do what it is supposed to do?")
- Typically this is done by calling the function, passing in predetermined parameters and checking the result.
- Example: `absolute_test.py`

```
def absolute(number):
    if (number < 0):
        result = number * -1
    else:
        result = number
    return(result)

# Test cases
print(absolute(-13))
print(absolute(7))
```

Expected results:
13
7

James Tam



Creating A Large Document

- Recall: When creating a large document you should plan out the parts before doing any actual writing.

Step 1: Outline all the parts (no writing)

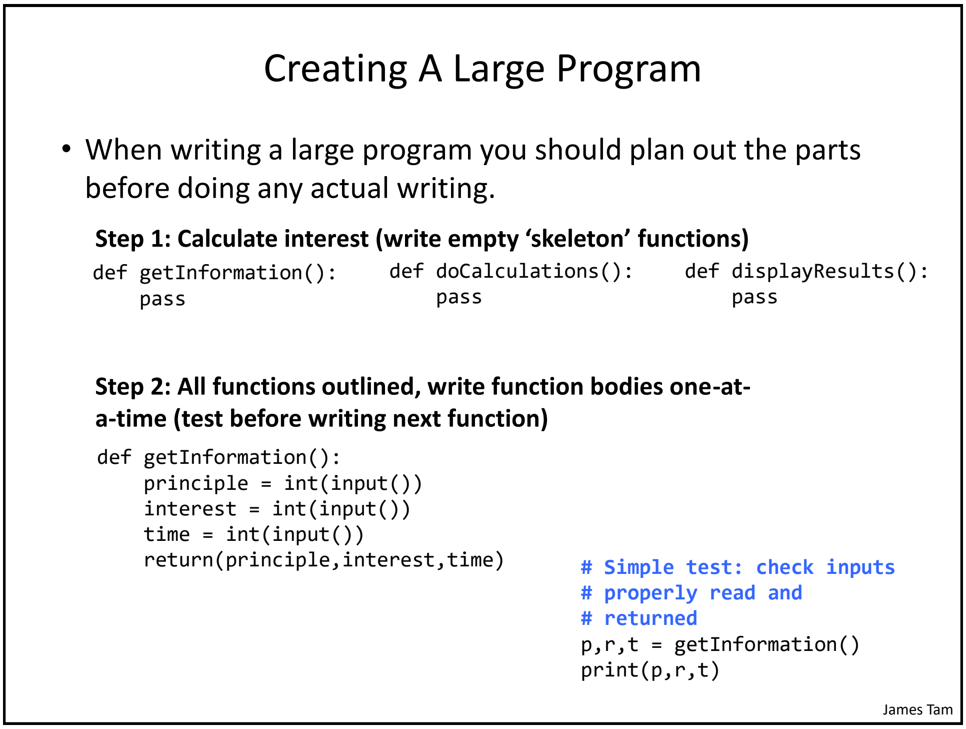
Chapter 1
- Introduction
- Section 1.1
- Section 1.2
- Section 1.3
- Conclusion

Chapter 2
- Introduction
- Section 2.1
- Section 2.2
- Section 2.3
- Section 2.4
- Conclusion

Chapter 3
- Introduction
- Section 3.1
- Section 3.2
- Conclusion

Step 2: After all parts outlined, now commence writing one part at a time

Section 1.1
It all started seven
and two score
years ago…



Creating A Large Program

- When writing a large program you should plan out the parts before doing any actual writing.

Step 1: Calculate interest (write empty 'skeleton' functions)

```
def getInformation():      def doCalculations():      def displayResults():
    pass                       pass                       pass
```

Step 2: All functions outlined, write function bodies one-at-a-time (test before writing next function)

```
def getInformation():
    principle = int(input())
    interest = int(input())
    time = int(input())
    return(principle,interest,time)
```

```
# Simple test: check inputs
# properly read and
# returned
p,r,t = getInformation()
print(p,r,t)
```

Why Employ Problem Decomposition And Modular Design (1)

- 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.

  - These are ‘one time’ costs: once you learn the basic principles of functions with one language then most languages will be similar.

Why Employ Problem Decomposition And Modular Design (2)

- 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:
    - Test one feature/function at a time
    - (Testing multiple features 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
- How and why to return values from a function
- What is a Boolean function
- What is the difference between a local and a global variable.
- How to document a function

James Tam

Copyright Notification

- "Unless otherwise indicated, all images in this presentation are used with permission from Microsoft."

James Tam