Functional Decomposition: Part 2

- Parameter passing
- Function return values
- Default arguments/optional parameters
- Return to globals: details why they should not be used.

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()

convert()
```

What is `celsius`???
What is `fahrenheit`???

Approx. scope

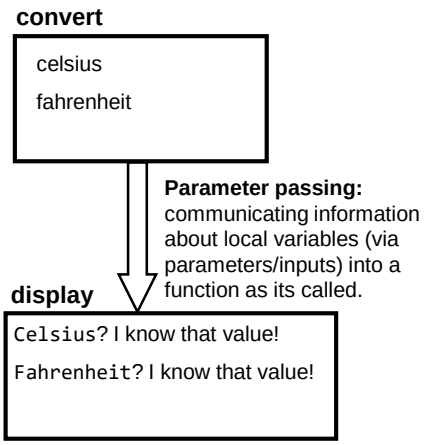
Variables `celsius` and `fahrenheit` are local to function `convert()`

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:



Synonyms for parameters
- Arguments
- Inputs

James Tam

New (Already Known?): Definitions

- **Function argument**: information that is passed into a function when that function is called.
- Examples you have seen with arguments:
  - `print("Hi-hi")`
  - `input("Name: ")`
  - `Random.randint(1,6)`
- Examples with different types:
  - `AVERAGE(2,6,4)`
  - `POWER(base,exponent)`
  - `CIRCUMEFRENCE(2,PI,radius)`
- Note: arguments can consist of **variables**, **named** and **unnamed constants**.

James Tam

Parameter Passing (Function Definition)

- List the names of the **variables in the round brackets that will store the name of the information** passed in.
- Functions can be defined with zero or more parameters.
  - The number of parameters should match between the function definition (examples bellow) and the function call.
- **Format:**

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

- **Example:**

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

James Tam

Parameter Passing (Function Call)

- Just list the information to be passed as **arguments** to the function in the round brackets.
  - Passing variables or named constant: just specify the name of the identifier e.g. `print(num)`, `random.ranrange(MIN,MAX)`
  - Passing unnamed constants: just specify the value of the parameter e.g. `input("Enter your name: ")`, `print("Hello")`
- **Format:**

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

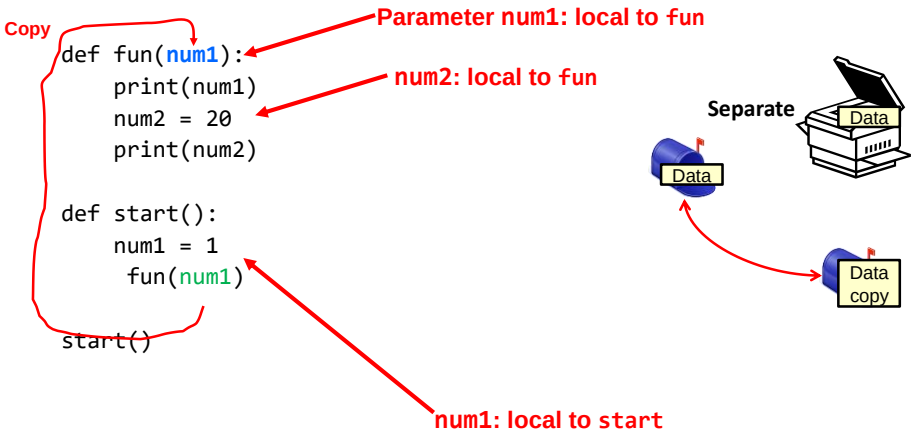
- **Example:**

```
display(celsius,fahrenheit)
```

James Tam

Memory And Parameter Passing

- **Parameters** passed as parameters/inputs into functions become **variables in the local memory** of that function.



James Tam

Arguments Vs. Parameters

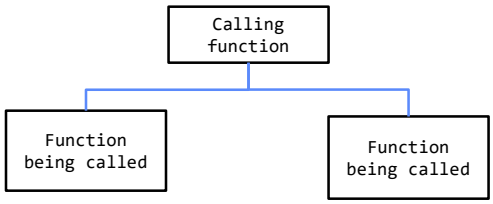
- **Arguments** are what is passed in during the function call.
- **Parameters** are the memory locations used to stored the arguments when the function is defined.
- Example:

```
#Defining what fun does it runs (after called)
def fun(name,age):
    print(name,age)

name=input()
age=int(input())
#Calling fun
fun(name,age)
```

James Tam

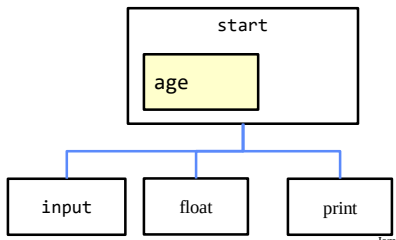
Structure Charts

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



- **Example**:

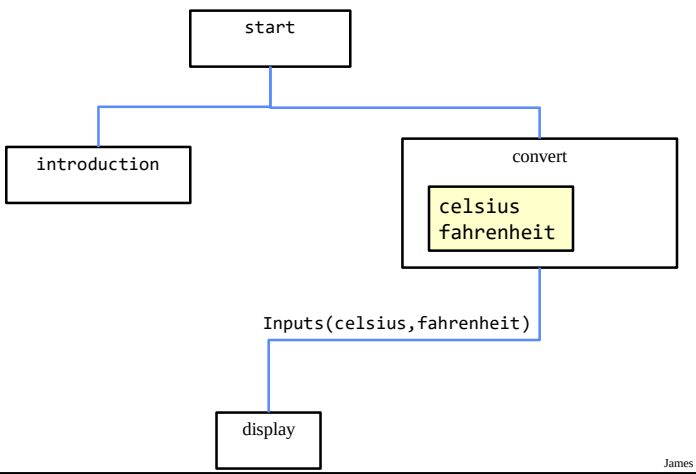
```
def start():
    age = float(input())
    print(age)
```



James Tam

Structure Chart: `temperature.py`

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



James Tam

Parameter Passing: Putting It All Together

- **Name of the example program**:
  `1temperature_parameters_arguments_inputs.py`
  - Learning objective: defining functions that take arguments/parameters/inputs when they are called.
  - Reminder: Function inputs does not refer to user input
    - e.g. `print("hello")` `#Input is the string hello`

```
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.
```

James Tam

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)


# Starting execution point
def start():
    introduction()
    convert()

start()
```

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

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

James Tam

Parameter Passing: Another Example

- **Name of the example program**: `2parameter_copy.py`
  - Learning objective: How function parameters/arguments/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()
```

Num1, num2 here are local to fun

Num1, num2 here are local to 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)

# Starting execution point
def start():
    num1 = 1
    num2 = 2
    str1 = "hello"
    fun1(num1,num2)
    fun2(num1,str1)

start()
```

DO THIS:

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()'

DO THIS:

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)

# starting exeuction point
def start():
    num1 = 1
    num2 = 2
    str1 = "hello"
    fun1(num1,num2)
    fun2(num1,str1)

start()
```

DON'T DO

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.

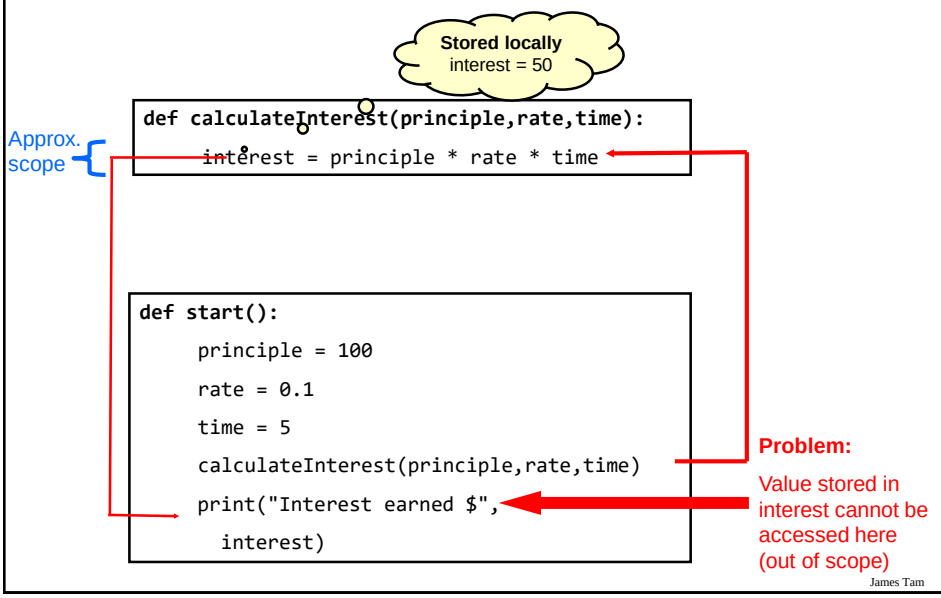
DON'T DO

Two numeric parameters are passed into the call for '`fun1()`' but only one parameter is listed in the definition for function '`fun1()`'

Good naming conventions can reduce incidents of this second type of mistake.

James Tam

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



James Tam

Solution: Have The Function Return Values Back To The Caller

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

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

Variable `interest` is still local to the function.

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**.

James Tam

Accessing Return Values

- Accessing means "to store" the values returned by a function.
- **Format (Single value returned)[1]:**

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

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

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

```
def calculateInterest():
    return(interest)          #Function definition

def display():
    interest = calculate(principle,rate,time)    #Function call
```

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.

James Tam

Accessing 'Multiple' Return Values

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

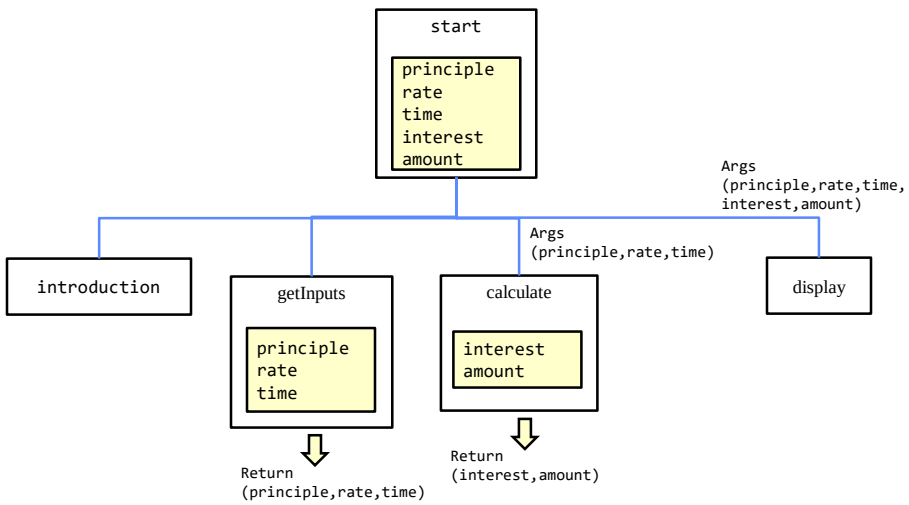
```
# Function definition
return(<value1>, <value 2>...)
# Function call
<variable 1>, <variable 2>... = <function name>()
```

- **Example ('Multiple' values returned):**

```
# Function definition
def getInputs():
    return(principle, rate, time)
# Function call
def start():
    principle, rate, time = getInputs()
```

James Tam

Structure Chart: interest.py



James Tam

Return Values: Putting It All Together

- **Name of the example program**: `3interest_return_values.py`
  - Learning objectives: 1) Returning values from a function you define # 2) Accessing values returned from a function you define.

```
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).
```

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)
```

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

Red: accessing/storing values returned from programmer defined functions.

Blue: programmer defined functions returning a value back to the caller.

James Tam

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
```

James Tam

Return Values: Putting It All Together (4)

```
# Starting execution point
def start():
    principle = -1
    rate = -1
    time = -1
    interest = -1
    amount = -1

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

start()
```

Red: accessing/storing values returned from programmer defined functions.

James Tam

Signifying The End Of A Function

- A function will immediately end and return back to the caller if:
  1. A return instruction is encountered (**return can be nothing"None"**)

```
def fun1():
    return
```

  2. End of the instructions in the function reached (no more indenting). **Return type is also "None".**

```
def fun2():
    print("hi hi")
    print() # Implicit return to caller (last instruction)
```

  3. For completeness: you can return something back that something is 'empty'. (More on the type of this empty memory location under 'composites')

```
def fun3():
    return()
```

James Tam

What Type Of Information Is Retuned Without A Return?

- **Name of the example program**: `4_function_return_types.py`
  - Learning objective: using the type function to specify the type of information, knowing the return type if nothing is returned.
  - The `type` function takes as an argument some data (variable, named or unnamed constant) and it returns the type of that data. E.g. int, float, bool, string, list

```
def fun1():
    print("fun1 return type: ",end="")

def fun2():
    print("fun2 return type: ", end="")
    return

def start():
    print(type(1),type(2/5),type(True),type("hi"),type([1,2]))
    print(type(fun1()))
    print(type(fun2()))
```

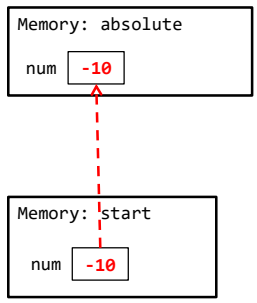
```
<class 'int'> <class 'float'> <class 'bool'> <class 'str'> <class 'list'>
fun1 return type: <class 'NoneType'>
fun2 return type: <class 'NoneType'>
```

James Tam

Parameter Passing Vs. Return Values

- Parameter passing is used to **pass information INTO a function** before the function executes (during the function call).
  - Parameters *are copied into variables* that are local to the function.

```
def absolute(num):
    etc.

def start():
    aStr = input("Enter number: ")
    num = int(aStr)
    absNum = absolute(num)
    print(absNum)
```

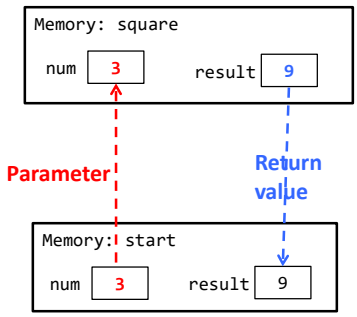


James Tam

Parameter Passing Vs. Return Values

- Return values are used to **communicate information OUT OF a function** as a function ends (going back/returning to a caller).
  - The return value must be stored in the caller of the function in the caller's local memory space.

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


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



James Tam

Default Arguments: Making Some Parameters Optional

- If the values for some parameters are not always known when the function is called then default values ("default arguments") can be specified.
- **New term, default argument**: if an argument is omitted then the default value is used for the missing value.
  - Thus the arguments which are provided with a default value are deemed as "optional arguments"
  - Syntax requirement:
    - Default arguments cannot be followed by non-default arguments.
    - Alternative wording: all optional arguments must be at end of the parameter list.
- **Format:**

```
def <function name>(<non default arg>, <non default arg>,...
  <default arg>=value>, <default arg=value>...):
```

- **Example:**

```
def display(studentID,anAge=-1,aName="No name"):
```

James Tam

Default Arguments: An Example

- **Name of the example program**: `5_default_parameters`
  - Learning objective: how define and call a function with default arguments.

```
def display(studentID,anAge=-1,aName="No name"):
    print(aName,studentID,anAge)

"""
Won't work: non-default argument'studentID' follows default
argument 'aName'
def wrongDisplay(aName="No name",studentID,anAge=-1,):
    print(aName,studentID,anAge)
"""

def start():
    display(123456,18,"Smiley")
    display(111111)
```

James Tam

Recall: In General Parameter Order Is Critical

```
def fun1(y,x):
    print(y,x)

def fun2(a,b):
    print(a,b)

#Starting execution point
def start():
    x = 1
    y = 2
    z = "hello"
    fun1(x,y)
    fun2(x,z)

start()
```

Order of the parameters (not the names) determines how parameters match

James Tam

New: Key Word Arguments

- The except is when the parameters being passed into a function are assigned values during the call.
  - In this case it's the name of the parameters that determine which parameters match up on the function call vs. the function definition.
  - In this case the **names must match**.
  - But order is not relevant.
- **Format (during function call):**
  <function name>(*<arg1>*=*<value>*, *<arg2>*=*<value>*, *<arg3>*=*<value>*…)
- **Example (during function call):**
  - `fun1(aNum=888,aStr="Lucky")`

James Tam

Illustrating Keyword Arguments

- **Name of the example program**: `6_keyword_arguments`
  - Learning objective: how to use keyword arguments (using the name of the arguments to specify how arguments are matched up in the function call vs. the function definition).

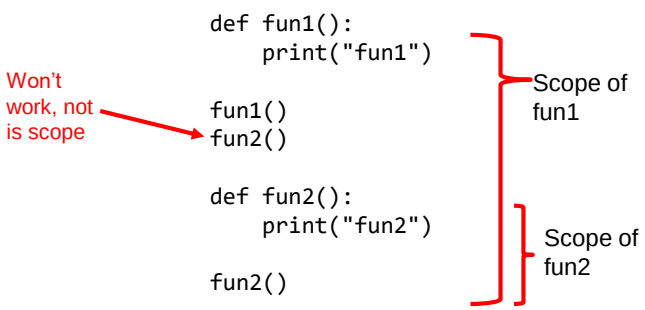
```
def fun1(aStr,aNum):
    print(aNum,aStr)

def fun2(localNum,localString):
    print(localNum,localString)

def fun3(aNum1,aStr1):
    print(aNum1,aStr1)

def start():
    fun1(aNum=888,aStr="Lucky")
    aNum=777
    fun2(aNum,"Also lucky")
fun3(aNum=888,aStr="lucky")
```

```
fun1(), using keyword arguments: 888 Lucky
fun2(), using normal positional arguments: 777 Also lucky
fun3(), using keyword arguments but names don't match:
```

James Tam

Function Scope

- Similar to variables and constants functions have a scope.
  - **New term, name space:** it's the part of the program where the function can be called i.e. it's scope.



James Tam

Recall: Program Design

- 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():
```

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
# are properly read as input
# and returned to caller
p,r,t = getInformation()
print(p,r,t)
```

James Tam

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

Solution When Outlining Your Program By Starting With 'Empty' Functions

```
def fun():
    print()

# Program'sstart
fun()
```

A function must have at least one instruction in the body

Alternative (writing an empty function: 'pass' a python instruction that literally does nothing)

```
def fun():
    pass

# Program's start
fun()
```

James Tam

Back To 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, a local is created).

```
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 changes in the function will refer to the global variable rather than creating a local one.

```
global <variable name>
```

James Tam

Globals: Another Example ('Write' Access Via The "Global" Keyword)

- **Name of the example program**: `7modifyingGlobals.py`
- Learning objective: How global variables can be modified inside functions.

```
num = 1

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

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

start()
```

References to the name 'num' in this function now affect the global variable, local variable not created inside function 'fun'

2 Global changed

1 Global

2 Global still changed after 'fun()' is done

James Tam

Why Define Variables Locally Rather Than Locally

- Benefits (why create them this way)
  - 1st: more efficient use of memory
  - **2nd: minimize the occurrence of side effects of global variables (last example)**
    - This is the main reason why it's regarded as bad style in actual practice.
    - But details are more complex so the explanation will come later.
      - The python `global` keyword allows global variables to be changed anywhere in the program.
      - This is another example of *tightly coupling code* as changes in one function may affect the rest of the program.
      - **You should not declare variables globally (and not use the `global` keyword).**
  - 3rd: pedagogical (creating variables locally forces you to apply important programming concepts such as parameter passing, function return values and scope).

James Tam

Students-Do: Practice Exercises (Functions)

- **Exercise 1** (very basic): write a function that takes a number as an argument and it doubles the number.
- **Exercise 2** (basic): write a function that takes a number as an argument and returns the absolute value of that number.
- **Exercise 3** (basic): write a function that takes two numbers as arguments and the quotient of the first argument divided by the second argument. If the second argument is zero then the division will not be performed and an error code of -1 will be returned.
- **Exercise 4** (basic): write a function that takes two numbers as arguments and it returns the lesser of the two.
- **Exercise 5** (basic-intermediate): write a function that takes three numbers as arguments and it returns the smallest.

James Tam

Students-Do: Practice Exercises (Functions Combined With Strings)

- **Exercise 6** (basic-intermediate): write a function that takes a string as argument and it returns a count of the number of vowels and consonants.
- **Exercise 7** (intermediate): write a function that takes a sentence as an argument and it returns a count of the number of words in that sentence.

James Tam

Students-Do: Practice Exercises (Functions Combined With Basic (1D) Lists)

- **List operations:** write functions that will perform these tasks. Each task should be implemented with it's own function.
  - Function takes a list of numbers as an argument and it returns the largest element.
  - Function takes a list of numbers as an argument and it returns the average of the list.
  - Function takes a list of strings as an argument and it a modified version of the list where all lower case alphabetic characters are converted to capitals.
  - Function takes a list (any type of elements) and it returns a new list which has reversed the order of the list passed as an argument.
  - Function takes a list of numbers as an argument and it returns the mode of the list (this one is harder than the rest but is a good challenge for developing your problem solving skills).

James Tam

Students-Do: Practice Exercises (Functions Combined With Basic (1D) Lists): 2

- **Writing Boolean functions**: returns a true or false value depending upon the characteristic of the argument.
  - Implement function '`isOdd`' that takes a list of numbers as an argument and determines if all the numbers are all odd.

James Tam

After This Section You Should Now Know

- How to pass information to functions via parameters
- How and why to return values from a function
- New definitions and concepts: function arguments, parameters, inputs, default arguments, keyword parameters.
- The use and effect of a pass instruction.
- How the global keyword works and why it should not be used.

James Tam

Copyright Notification

- Unless otherwise indicated, all images in this presentation were provided courtesy of James Tam.

James Tam