Functions: Decomposition And Code Reuse, Part 2

- Parameter passing
- Function return values
- Function specific style requirements (rules of thumb for good style)
- Documenting functions

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`???

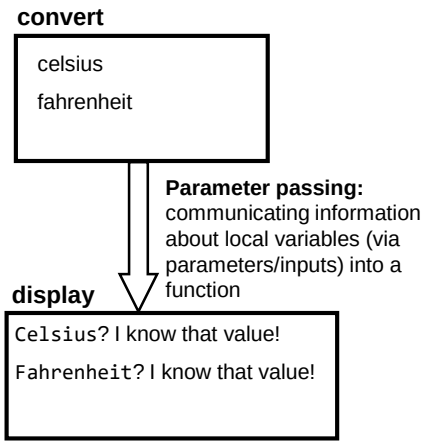
Variables `celsius` and `fahrenheit` are local to function `convert()`

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 parameter/argument/input**: information that is passed into a function when that function is called.
- Examples:
  - `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.
- **Format:**

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

- **Example:**

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

n is a non-negative integer (you pass in 0 or more parameters)

James Tam

Parameter Passing (Function Call)

- Just list the information to be passed as parameters 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>)
```

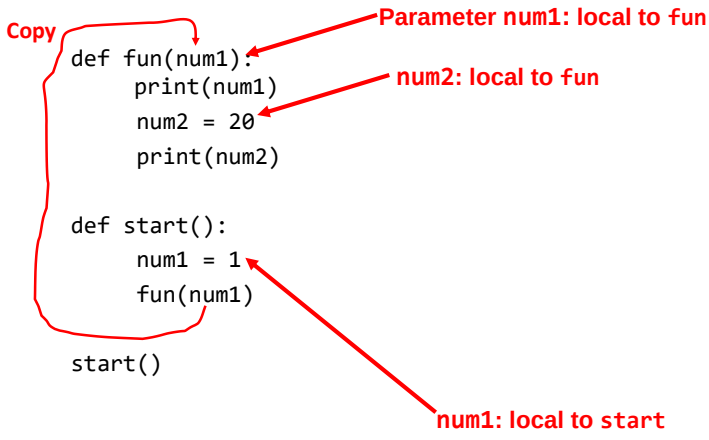
- **Example:**

```
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

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

James Tam

Sample (Simple) Example Question: Terminology

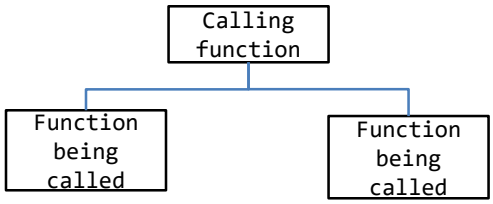
- Write a function that takes **two inputs**: a numerator and denominator
- The function will calculate and display onscreen the floating point quotient
- Solution:
  - There is no mention of *user* input
    - Don't call the `input()` function!
  - Consequently the input in the program description refers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

```
def displayQuotient(numerator,denominator): #Correct
    quotient = numerator/denominator
    print(quotient)

def displayQuotient(): #Incorrect
    numerator = float(input())
    ...
```

James Tam

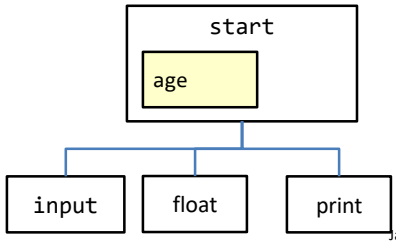
Structure Charts

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



- **Example**:

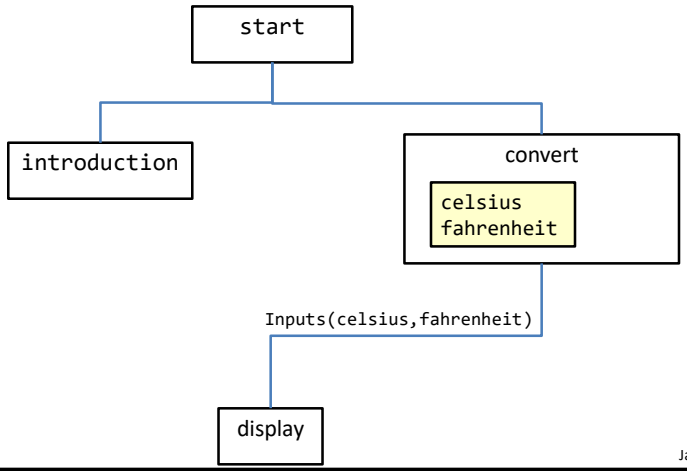
```
def start():
    age = float(input())
    print(age)
```



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**:
  `4temperature_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

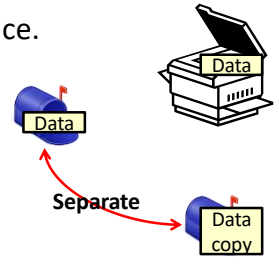
Parameter Passing: Important Recap!

- A parameter is copied into a local memory space.

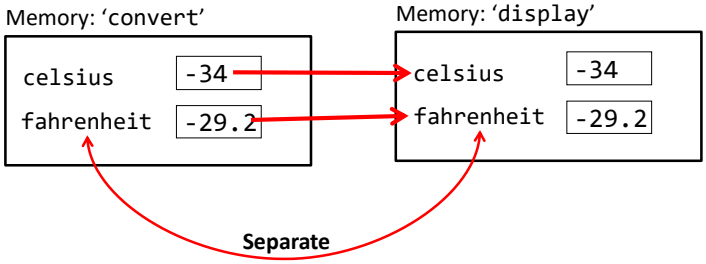
```
# Inside function convert()
display(celsius,fahrenheit)     # Function call

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

Make copy

Make copy



RAM



James Tam

Parameter Passing: Another Example

- **Name of the example program**: `5parameter_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()
```

James Tam

A 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 (incorrect)
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()
```

James Tam

Parameter Passing: Why It's Needed

- 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!
```

A Common Mistake: Forgetting To Pass Parameters

- Example:

```
def displayAge():
    print(age)


def start():
    age = int(input("Age in years: "))
    displayAge()
```

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.

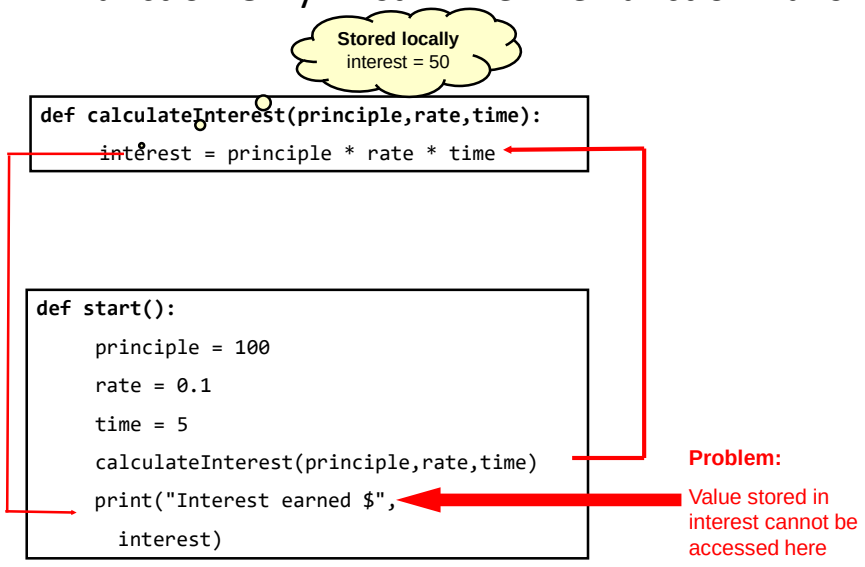
James Tam

Scope: A Variant Example

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

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

fun2()
```

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

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



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".

Why Are Function Return Values Needed?

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

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

RAM

Memory: 'calculateArea'

w	
l	
a	

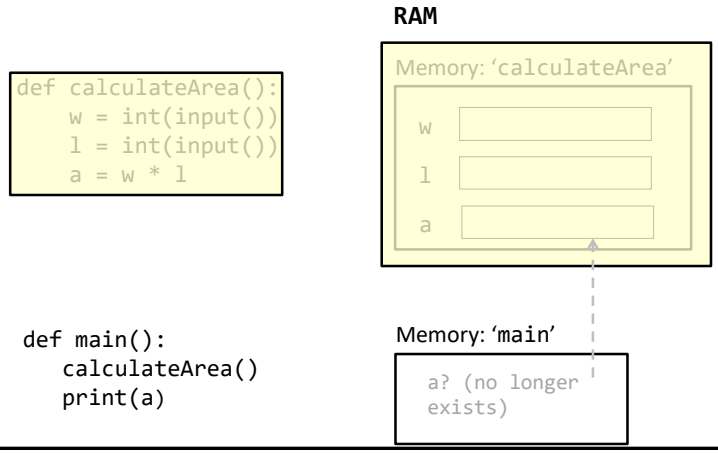
```
def main():
   calculateArea()
   print(area)
```

Memory: 'main'

James Tam

Why Are Function Return Values Needed? (2)

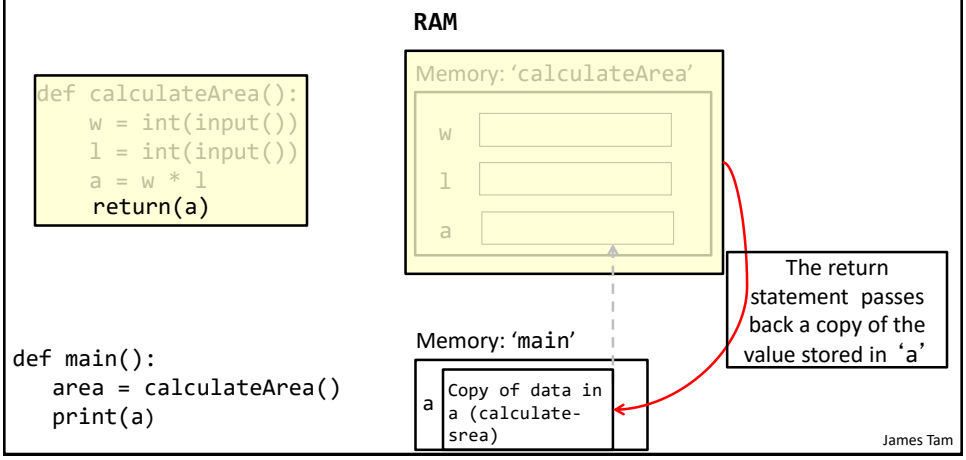
- After a function has ended local variables are 'gone'.



James Tam

Why Are Function Return Values Needed? (3)

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



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)

def display():
   interest = calculate(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.

Accessing Return Values

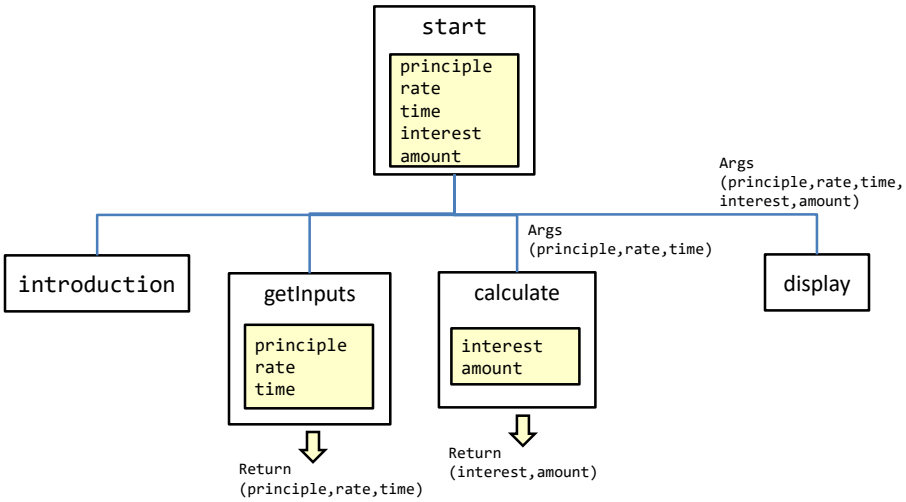
- **Format (Multiple values returned):**

```
# 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()
```

Structure Chart: `interest.py`



James Tam

Accessing Return Values: Putting It All Together

- **Name of the example program**: `6interest_return_values.py`
  - Learning objective: parameter passing, defining functions with parameters passed in when they are called and return outputs when they end.

```
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).
```

Accessing 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)
```

Accessing 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
```

Accessing Return Values: Putting It All Together (4)

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

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

start()
```

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 empty "None")

```
def convert(catAge):
    if (catAge < 0):
        print("Can't convert negative age to human years")
        return()    # Explicit return to caller (return
                    # instruction)
    else:
        :  :
```

2. There are no more instructions in the function (no more indenting).

```
def introduction():
    print()
    print("TAMCO INC. Investment simulation program")
    print("All rights reserved")
    print() # Implicit return to caller (last instruction)
```

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** so they can be used)

```
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.
- Function return values must be explicitly saved by the caller of the function.
- **Name of the example program**: `7mistakeSavingReturn.py`

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

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

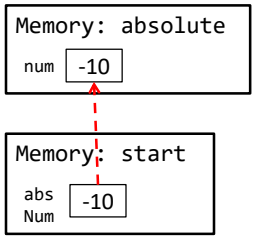
```
# Start: error
area = 0
calculateArea(4,3)
print(area)
```

```
# 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** 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)
```

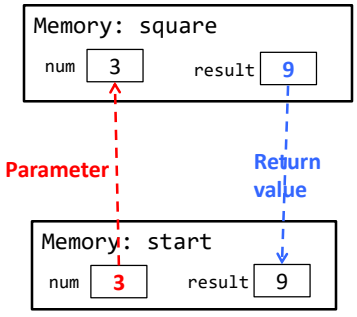


James Tam

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.

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


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



James Tam

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.

James Tam

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: Asks a question**
   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.
   b) See each assignment description for what constitutes "one screen".
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()`

   (Python doesn't follow this convention but it's an exception).

James Tam

Documenting Functions (Parameters)

- 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

Documenting Functions (Return Values)

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

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

James Tam

Documenting Functions

- (As previously mentioned the documentation should include)
  - The type and number of parameters/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:
    - List of program features implemented in a function should be specified in the documentation for that function.
    - 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

After This Section You Should Now Know

- How to pass information to functions via parameters
- How and why to return values from a function
- How to document a function
- Function specific style requirements
- New definitions: function arguments, parameters, inputs

James Tam

Copyright Notification

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

James Tam