# CS303E: Elements of Computers and Programming

Functions

Dr. Bill Young
Department of Computer Science
University of Texas at Austin



Last updated: August 27, 2024 at 14:25



## Functions

You probably remember functions from your high school math classes:

$$f(x) = x^2 + 2$$

This defines a recipe for performing a computation. It has a parameter $x$, which doesn't have a value but stands for any number you want to put there.

Notice that the definition *doesn't perform* a computation. It only tells you how to perform one. Namely, given any specific value for $x$, square it and add 2.

Thus, $f(5)$ equals 27, because (substituting 5 for $x$ in our rule),

$$f(5) = 5^2 + 2 = 27$$



## Functions

Notice several things:

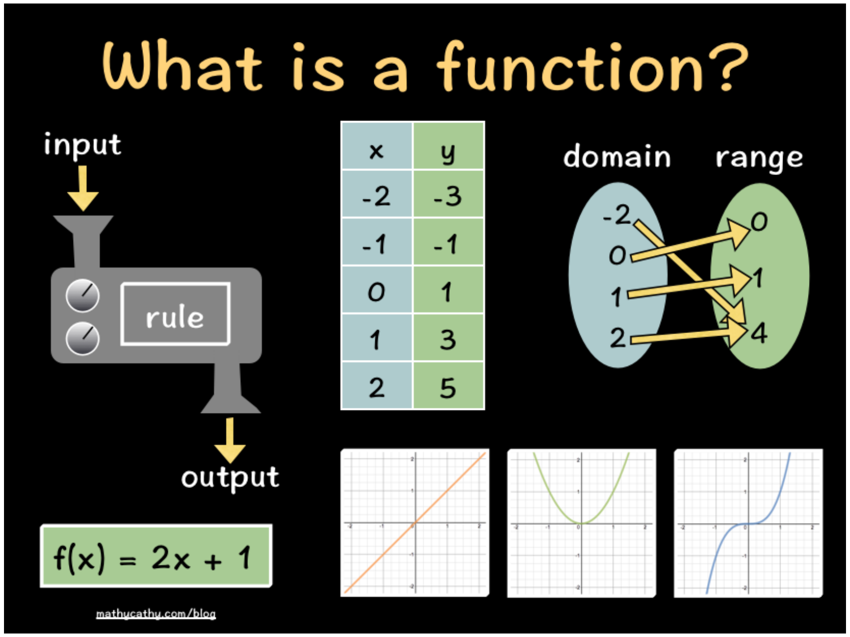
- The *function definition* doesn't perform a computation; it only gives us a recipe or procedure for performing a computation.
- A *function call* (e.g., $f(5)$) does perform a computation, using the defining rule.
- To actually compute something, we need to *call the function*, supplying values for the parameters.
- The computed value is "returned" to the calling environment replacing the call with the value.

$$f(3) + f(2) = (3^2 + 2) + (2^2 + 2) = 11 + 6 = 17$$

Functions in programming languages work similarly, with a few differences.



## What is a Function?



# User-Defined Functions

We've seen lots of system-defined functions; now it's time to define our own.

General form:

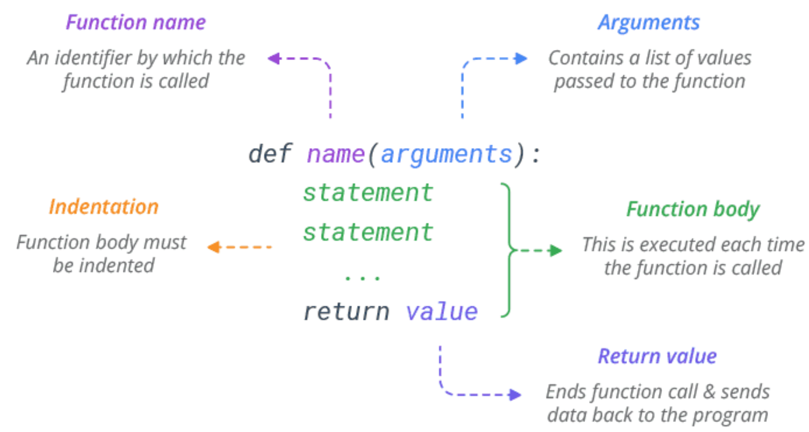
```
def functionName( list of parameters ):   # header
   statement(s)                           # body
```

**Meaning:** a function definition defines a block of code that performs a specific task. It can reference any of the variables in the list of parameters. It may or may not return a value.

The parameters are **formal parameters**; they stand for arguments passed to the function later.

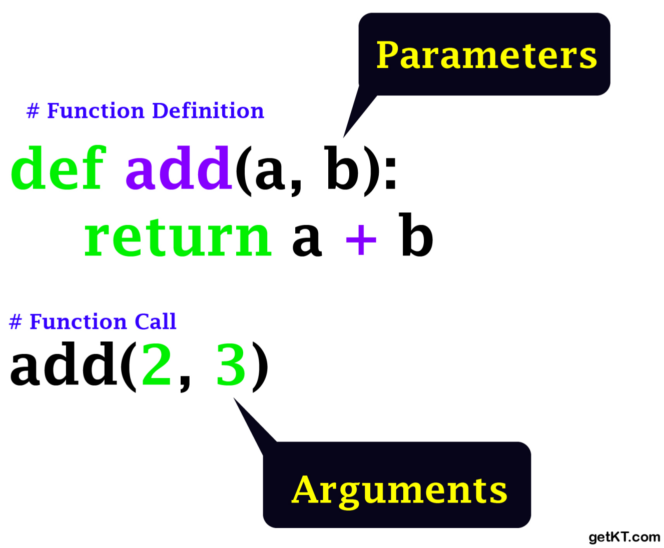


# Functions






# Calling a Function






# Function Example

Suppose you want to sum the integers 1 to n.

In file `functionExamples.py`:

```
def sumToN( n ):
    """ Sum the integers from 1 to n. """
    sum = 0                          # identity element for +
    for i in range(1, n + 1 ):       # Why n+1?
        sum += i
    # Hand the answer to the calling environment.
    return sum
```

## Function Example

Notice that the definition defines a *function* to perform the task, but *doesn't actually perform the task*. We still have to call/invoke the function with specific arguments.

```
>>> from functionExamples import *
>>> sumToN( 10 )
55
>>> sumToN( 1000 )
500500
>>> sumToN(1000000)
500000500000
```

You can think of the function call as being *replaced by* the value returned.

You can call a function as many times as you need.



## Some Observations

```
def sumToN( n ):              # function header
    ....                      # function body
```

Here n is a *formal parameter*. It is used in the definition as a place holder for an *actual parameter* (e.g., 10 or 1000) in any specific call.

For a specific value of x, sumToN(x) *returns* an int value, meaning that a call to sumToN can be used anyplace an int expression can be used.

```
>>> print( sumToN( 50 ) )
1275
>>> ans = sumToN( 30 )
>>> print( ans )
465
>>> print( "Even" if sumToN(3) % 2 == 0 else "Odd" )
Even
```



## Functional Abstraction

Once we've defined sumToN, we can use it almost as if were a primitive in the language without worrying about the details of the definition.

*We need to know what it does, but don't care anymore how it does it!*

This is called **information hiding** or **functional abstraction**.



## Another Way to Add Integers 1 to N

Suppose later we discover that we could have coded sumToN more efficiently (as discovered by the 8-year old C.F. Gauss in 1785):

```
def sumToN( n ):
    """ Sum the integers from 1 to n. """
    return ( n * (n+1) ) // 2
```

*Because we defined* sumToN *as a function, we can just swap in this definition without changing any other code.* If we'd done the implementation in-line, we'd have had to go find every instance and change it.

```
>>> sumToN(10)
55
>>> sumToN( 1000000000000 )
500000000000500000000000
```

## Return Statements

When you execute a `return` statement, you go back to the calling environment. You may or may not hand a value back to the caller.

General forms:

```
return
return expression
```

A `return` that doesn't specify a value actually returns the constant `None`.

Every function has an *implicit* return at the end.

```
def printTest ( x ):
  print( x )
  # implicit return here
```

A `return` statement can only appear within a function, but you can have as many as you need.



## Return vs. Print

You can think of a Python function as a recipe for performing some computation. The computation *may* return a value:

```
def cube( x ):
    return x ** 3
```

If it does, you can think of the value returned as replacing the call to the function.

```
    cube( 3 ) + cube ( 2 )
```

is the same as

```
    27 + 8
```



## Return vs. Print

But often a Python function doesn't return a value (actually returns the constant None) either always or sometimes.

```
def cubeNonnegative( x ):
    if x >= 0:
       return x ** 3
    else:
       print( "Negative argument supplied." )
```

Notice that this returns an int value for nonnegative arguments but returns None and prints an error message for negative arguments.

**Almost always print an error message; don't return it.** A caller expecting an int value to be returned probably can't handle a string.

The point of an error message is to inform the user that something went wrong.



## Remember: The Framework of a Simple Python Program

Define your program in file `Filename.py`:

```
def main ():

   Python statement
   Python statement
   Python statement
       ...
   Python statement
   Python statement
   Python statement

main ()
```

To run it:

```
> python Filename.py
```

This defines a function `main`; could have a different name.

If you wanted to end the program, you could include a `return` statement.

You couldn't use `return` if you just had the statements at the top level.

## About return

In file `returnExamples.py`:

```
def printSquares():
    """ Compute and print squares until 0 is entered
        by the user. """
    while True:
        num = int(input("Enter an integer or 0 to exit: "))
        if ( num != 0 ):           # "if num:" works
            print( "The square of", num, "is:", num ** 2 )
        else:
            return                 # no value is returned

printSquares()
```

This doesn't return a value, but accomplishes it's purpose by the "side effect" of printing.



## About return

```
> python returnExamples.py
Enter an integer or 0 to exit: 7
The square of 7 is: 49
Enter an integer or 0 to exit: -12
The square of -12 is: 144
Enter an integer or 0 to exit: 0
>
```

A function that "doesn't return a value" actually returns the constant None.



## Some More Function Examples

Suppose we want to multiply the integers from 1 to n:

```
def multToN( n ):
    """ Compute the product of the numbers from 1 to n. """
    prod = 1                  # identity element for *
    for i in range(1, n+1):
        prod *= i
    return prod
```

Convert fahrenheit to celsius:

```
def fahrToCelsius( f ):
""" Convert fahrenheit temperature value to celsius
    using formula: C = 5/9(F-32). """
    return 5 / 9 * ( f - 32 )
```

Or celsius to fahrenheit:

```
def celsiusToFahr( c ):
    """ Convert celsius temperature value to fahrenheit
        using formula: F = 9/5 * C + 32. """
    return 9 / 5 * c + 32
```



## Fahr to Celsius Table

In slideset 1, we showed the C version of a program to print a table of Fahrenheit to Celsius values. Here's a Python version:

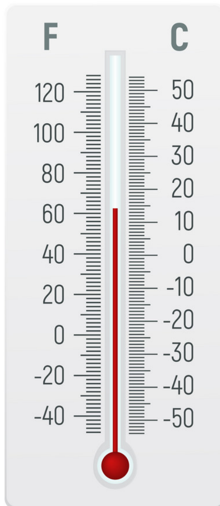
In file `FahrToCelsius.py`:

```
from functionExamples import fahrToCelsius

def printFahrToCelsius():
    """ Print table fahrenheit to celsius for temp
        in [0, 20, 40, ... 300]."""
    lower = 0
    upper = 300
    step  = 20
    print("Fahr\tCelsius")
    for fahr in range( lower, upper + 1, step ):
        # Use an already defined function.
        celsius = fahrToCelsius( fahr )
        print( format( fahr, "3d"), "\t", \
               format( celsius, "6.2f") )
    return          # not actually necessary

printFahrToCelsius()
```

Notice that `printFahrToCelsius` returns None.

# Running the Temperature Program

```
> python FahrToCelsius.py
Fahr     Celsius
  0       -17.78
 20        -6.67
 40         4.44
 60        15.56
 80        26.67
100        37.78
120        48.89
140        60.00
160        71.11
180        82.22
200        93.33
220       104.44
240       115.56
260       126.67
280       137.78
300       148.89
```




**Exercise:** Do a similar problem converting Celsius to Fahrenheit.



# Functions Inside Functions

There are occasionally reasons to define one function within another function. *But it's generally a bad idea, unless you know what you're doing.*

**For this class, always define your functions at the top level of your .py file.**



# Let's Take a Break




# A Bigger Example: Print First 100 Primes

Suppose you want to print out a table of the first 100 primes, 10 per line.

You could sit down and write this program from scratch, without using functions. But it would be a complicated mess (see section 5.8 of the book).

Better to use **functional abstraction**: find parts of the algorithm that can be coded separately and "packaged" as functions.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 5 | 7 | 11 | 13 | 17 | 19 | 23 | 29 |
| 31 | 37 | 41 | 43 | 47 | 53 | 59 | 61 | 67 | 71 |
| 73 | 79 | 83 | 89 | 97 | 101 | 103 | 107 | 109 | 113 |
| 127 | 131 | 137 | 139 | 149 | 151 | 157 | 163 | 167 | 173 |
| 179 | 181 | 191 | 193 | 197 | 199 | 211 | 223 | 227 | 229 |
| 233 | 239 | 241 | 251 | 257 | 263 | 269 | 271 | 277 | 281 |
| 283 | 293 | 307 | 311 | 313 | 317 | 331 | 337 | 347 | 349 |
| 353 | 359 | 367 | 373 | 379 | 383 | 389 | 397 | 401 | 409 |
| 419 | 421 | 431 | 433 | 439 | 443 | 449 | 457 | 461 | 463 |
| 467 | 479 | 487 | 491 | 499 | 503 | 509 | 521 | 523 | 541 |

## Print First 100 Primes: Algorithm

Here's some Python-like pseudocode to print 100 primes:

```
def print100Primes():
    primeCount = 0
    num = 0
    while (primeCount < 100):
       if (we've already printed 10 on the current line):
          go to a new line
       nextPrime = ( get the next prime > num )
       print nextPrime on the current line
       num = nextPrime
       primeCount += 1
```

Note that most of this is just straightforward Python programming! The only "new" part is how to find the next prime. So we'll make a *function*.



## Top Down Development

So let's *assume* we can define a function:

```
def findNextPrime( num ):
    """ Return the first prime greater
        than num. """
    < body >
```

in such a way that it returns the first prime larger than `num`.

Is that even possible? Is there always a "next" prime larger than num?



## Top Down Development

So let's *assume* we can define a function:

```
def findNextPrime( num ):
    """ Return the first prime greater
        than num. """
    < body >
```

in such a way that it returns the first prime larger than `num`.

Is that even possible? Is there always a "next" prime larger than num?

Yes! There are an infinite number of primes. So if we keep testing successive numbers starting at `num + 1`, we'll eventually find the next prime. *That may not be the most efficient way!*



## Value of Functional Abstraction

Notice we're following a "divide and conquer" approach: Reduce the solution of our bigger problem into one or more subproblems which we can tackle independently.

It's also an instance of "information hiding." We don't want to think about how to find the next prime, while we're worrying about printing 100 primes. Put that off!

## Next Step

Now solve the original problem, *assuming* we can write `findNextPrime`.

In file `IsPrime3.py`:

```
def print100Primes ():
   """ Print a table of the first 100 primes,
       10 primes per line. """
   primeCount = 0                 # primes we've found
   onLine = 0                     # primes printed on line
   num = 0                        # need next prime > num
   while (primeCount < 100):
      # Do we stay on current line?
      if ( onLine >= 10 ):
         print()
         onLine = 0
      # This is the only thing left to define:
      nextPrime = findNextPrime( num )
      num = nextPrime
      primeCount += 1
      print( format( nextPrime, "3d"), end = " " )
      onLine += 1
   print()
```



## Looking Ahead

Here's what the output should look like.

```
>>> from IsPrime3 import print100Primes
>>> print100Primes()
  2   3   5   7  11  13  17  19  23  29
 31  37  41  43  47  53  59  61  67  71
 73  79  83  89  97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229
233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349
353 359 367 373 379 383 389 397 401 409
419 421 431 433 439 443 449 457 461 463
467 479 487 491 499 503 509 521 523 541
```

Of course, we couldn't do this if we really hadn't defined `findNextPrime`. So let's see what that looks like.



## How to Find the Next Prime

The next prime (> num) can be found as indicated in the following pseudocode:

```
def findNextPrime( num ):
    if num < 2:
       return 2 as the answer
    else:
       guess = num + 1
       while ( guess is not prime )
           guess += 1
       return guess as the answer
```

Again we solved one problem by assuming the solution to another problem: deciding whether a number is prime.

Can you think of ways to improve this algorithm?



## Here's the Implementation

Note that we're assuming we can write:

```
def isPrime( num ):
    """ Boolean test for primality. """
    < body >
```

```
def findNextPrime( num ):
   """ Find the first prime > num. """
   if ( num < 2 ):
      return 2
   guess = num + 1
   while ( not isPrime( guess ) ):
      guess += 1
   return guess
```

This works (assuming we can define `isPrime`), but it's pretty inefficient. How could you fix it?

## Find Next Prime: A Better Version

When looking for the next prime, we don't have to test every number, just the odd numbers (after 2).

```
def findNextPrime ( num ):
   """ Find the first prime > num. """
   if ( num < 2 ):
      return 2

   # If (num >= 2 and num is even), the
   # next prime after num is at least
   # (num - 1) + 2, which is odd.
   if (num % 2 == 0):
      num -= 1
   guess = num + 2

   while ( not isPrime( guess )):
      guess += 2
   return guess
```

Now all that remains is to write `isPrime`.



## Is a Number Prime?

We already solved a version of this in slideset 5. Let's rewrite that code as a Boolean-valued function:

```
def isPrime( num ):
   """ Test whether num is prime. """

   # Deal with evens and numbers < 2.
   if (num < 2 or num % 2 == 0 ):
      return ( num == 2 )          # Why this value?

   # See if there are any odd divisors
   # up to the square root of num.
   divisor = 3
   while (divisor <= math.sqrt( num )):
      if ( num % divisor == 0 ):
         return False
      else:
         divisor += 2
   return True
```

By the way, a Boolean-valued function is often called a **predicate**.



## Testing Our Code

```
>>> from IsPrime3 import findNextPrime, isPrime
>>> findNextPrime( -10 )
2
>>> findNextPrime( 2 )
3
>>> findNextPrime( 1000 )
1009
>>> findNextPrime( 100000000 )
100000007
>>> isPrime( 100000007 )
True
>>> isPrime( 1001 )
False
>>> isPrime( 1003 )
False
>>> isPrime( 1007 )
False
>>> isPrime( 1009 )
True
```



## One More Example

Suppose we want to find and print k primes, starting from a given number:

In file `IsPrime3.py`:

```
def findKPrimesStartingFrom ( k, num ):
   """ Find the next k primes bigger than num. """
   if (k < 1):
      print( "You asked for zero primes!" )
   else:
      for i in range( k ):
         nextPrime = findNextPrime( num )
         print( nextPrime, end=" " )
         num = nextPrime
      print()
```

Notice that we can use functions we've defined such as `findNextPrime` and `isPrime` (almost) *as if* they were Python primitives.

## Running Our Program

```
>>> from IsPrime3 import findKPrimesStartingFrom
>>> findKPrimesStartingFrom( -10, 100000000 )
You asked for zero primes!
>>> findKPrimesStartingFrom( 5, -10 )
2 3 5 7 11
>>> findKPrimesStartingFrom( 10, 100000000 )
100000007 100000037 100000039 100000049 100000073 100000081
    100000123 100000127 100000193 100000213
```



## Functions and Return Values

Functions can return a value or not. A function that doesn't return a value is sometimes called a **procedure**.

Of the functions defined earlier:

- `sumToInt`, `multToN`, `findNextPrime` all return `int` values
- `farhToCelsius` and `celsiusToFahr` return `float` values
- isPrime returns a `bool` value
- `printSquares`, `printFahrToCelsius`, `print100Primes`, and `findKPrimesStartingFrom` don't return a value (return `None`).



## Let's Take a Break




## Positional Arguments

This function has four formal parameters:

```
def functionName ( x1, x2, x3, x4 ):
    < body >
```

Any call to this function should have exactly four actual arguments, which are matched to the corresponding formal parameters:

```
    functionName( 9, 12, -3, 10 )
    functionName( 'a', 'b', 'c', 'd' )
    functionName( 2, "xyz", 2.5, [3, 4, 5] )
```

This is called using **positional** arguments; it's by far the most common approach.

## Keyword Arguments

It is also possible to use the formal parameters as **keywords**.

```
def functionName ( x1 , x2 , x3 , x4 ):
    functionBody
```

These two calls are equivalent:

```
    functionName( 'a', 'b', 'c', 'd' )
    functionName( x3 = 'c', x1 = 'a', x2 = 'b', x4 = 'd' )
```

You can list the keyword arguments in any order, but all must still be specified.



## Mixing Keyword and Positional Arguments

You can mix keyword and positional arguments, but *must* have positional arguments first in order.

```
def functionName ( x1, x2, x3, x4 ):
   functionBody

functionName( 'a', 'b', x4 = 'd', x3 = 'c' )     # OK
functionName( x2 = 'b', x1 = 'a', 'c', 'd' )     # illegal
```

Why do you think they make this rule?



## Default Parameters

You can also specify **default arguments** for a function. If you don't specify a corresponding actual argument, the default is used.

```
def printRectangleArea( width = 1, height = 2 ):
    area = width * height
    print("width: ", width, "\theight: ", height, \
          "\tarea:", area)

printRectangleArea()                      # use defaults
printRectangleArea(4, 2.5)                # positional args
printRectangleArea(height = 5, width = 3) # keyword args
printRectangleArea(width = 1.2)           # default height
printRectangleArea(height = 6.2)          # default width
```



## Using Defaults

```
> python RectangleArea.py
width:  1         height:  2         area: 2
width:  4         height:  2.5       area: 10.0
width:  3         height:  5         area: 15
width:  1.2       height:  2         area: 2.4
width:  1         height:  6.2       area: 6.2
```

Notice that you can mix default and non-default arguments, but must define the non-default arguments first.

```
def email (address, message = ""):
```

## Passing by Reference

*All values in Python are objects, including numbers, strings, etc.*

When you pass an argument to a function, you're actually passing a **reference** (pointer) to the object, not the object itself.

There are two kinds of objects in Python:

- mutable: you can change them in your program.
- immutable: you can't change them in your program.

*If you pass a reference to a mutable object, it can be changed by your function. If you pass a reference to an immutable object, it can't be changed by your function.*



## Python Types

| Class | Description | Syntax example |
|---|---|---|
| int | An immutable fixed precision number of unlimited magnitude | 42 |
| float | An immutable floating point number (system-defined precision) | 3.1415927 |
| str | An immutable sequence of characters. | 'Wikipedia'<br>"Wikipedia"<br>"""Spanning multiple lines""" |
| bool | An immutable truth value | True, False |
| tuple | Immutable, can contain mixed types | (4.0, 'string', True) |
| bytes | An immutable sequence of bytes | b'Some ASCII'<br>b"Some ASCII" |
| list | Mutable, can contain mixed types | [4.0, 'string', True] |
| set | Mutable, unordered, no duplicates | {4.0, 'string', True} |
| dict | A mutable group of key and value pairs | {'key1': 1.0, 3: False} |



## Passing an Immutable Object

Consider the following code:

```
def increment (x):
  x += 1
  print( "Within the call x is: ", x )

x = 3
print( "Before the call x is: ", x )
increment( x )
print( "After the call x is: ", x )

def revList (lst):
  lst.reverse()
  print( "Within the call lst is: ", lst )

lst = [1, 2, 3]
print( "Before the call lst is: ", lst )
revList( lst )
print( "After the call lst is: ", lst )
```



## Passing Immutable and Mutable Objects

Invoking this code:

```
>python Test.py
Before the call x is:  3
Within the call x is:  4
After the call x is:  3

Before the call lst is:  [1, 2, 3]
Within the call lst is:  [3, 2, 1]
After the call lst is:  [3, 2, 1]
```

Notice that the immutable integer parameter to `increment` was unchanged, while the mutable list parameter to `revList` was changed by the call.

## Scope of Variables

Variables defined in a Python program have an associated **scope**, meaning the portion of the program in which they are defined.

A **global variable** is defined outside of a function and is visible after it is defined. *Use of global variables is generally considered bad programming practice. Don't use them unless you have a very good reason!*


A **local variable** is defined within a function and is visible from the definition until the end of the function.

A local definition overrides a global definition.



## Overriding

A local definition (locally) overrides the global definition.

```
x = 1                    # x is global

def func():
    x = 2                # this x is local
    print( x )           # will print 2

func()
print( x )               # will print 1
```

Running the program:

```
> python funcy.py
2
1
```



## Global Variables

```
callCount = 0          # global variable

def caller ():
  global callCount     # needed to access
  callCount += 1

caller()
print( "callCount = ", callCount )
caller()
print( "callCount = ", callCount )
caller()
print( "callCount = ", callCount )
```

```
> python Test.py
callCount =  1
callCount =  2
callCount =  3
```

What would happen if you took out the line containing `global`?



## Returning Multiple Values

The Python `return` statement can also return multiple values. Actually, it returns a *tuple* of values.

```
def multipleValues ( x, y ):
  return x + 1, y + 1

print( "Values returned are: ",  multipleValues ( 4, 5.2 ))

x1, x2 = multipleValues( 4, 5.2 )
print( "x1: ", x1, "\tx2: ", x2 )
```

```
Values returned are:  (5, 6.2)
x1:  5  x2:  6.2
```

You can operate on this using tuple functions, which we'll cover later in the semester, or assign them to variables.

# Order of Things in a .py File

Python is pretty permissive about the order of things in your .py file. The following is the order I prefer:

- Header / Extended comment explaining what's in the file
- Any imports required
- Program constants
- Function definitions
- `main` function definition
- Call to `main` function


Include comments throughout.




**Next stop:** Objects and Classes.