FUNCTIONS

CHAPTER 6

FUNCTION OVERVIEW

- We've used built-in functions:
  - Examples:

```
print("ABC", x+10, sep=":")
round(x * 5, 2)
pygame.draw.circle(screen, (255,0,0), (400,300), 10, 0)
```

  - Behind the scenes: many lines of python/C++
- Now we'll look at writing our own
  - A way to group lines of related code.
  - Examples:
    - A function to draw the main character
    - A function to calculate the distance between 2 points
    - A function to draw a game menu.

WHY USE FUNCTIONS?

- Readability (to you and others)
- Problem Decomposition
  - Team-based projects
- Code re-use (removing **redundancy**)
  - Make your own library (module)
- **Abstraction**
- **Encapsulation**

FUCNTION DEFINITION SYNTAX

```
def newFuncNoParams():
    """ A 'docstring' explaining the func"""
    # Put the body (1-n lines) here
    # (or a pass stmt., temporarily)

def newFuncWithParams(param1, param2, …):
    """ A 'docstring' explaining the func"""
    # Put the body (1-n lines) here
    # NOTE: the … above just means more params
    #       It's not valid python syntax
```

PARAMETERS

- A way to tell a function *how* to do its job.
  - Makes it useful in more situations.
- Example:

```
def f(a, b):
    """ Adds two numbers and prints"""
    print(a + b)
x = 9
f(4, x)
```

- Parameters vs. Arguments
  - **Arguments**: Values copied to parameters when a function is called
    - Can be constants, variables (the value is copied), expressions, another function call, etc.
  - **Parameters**: Variables used in the function
    - Created when the function starts; destroyed when it ends.

EXAMPLE

[A function to draw an ASCII box]

+ add Params

DEFAULT AND KEYWORD PARAMETERS

- Default params

```
def func(a, b, c=9, d=3):
      print(a + b + c + d)
func(1,2,3,4)    # Prints 10
func(1,2,5)      # Prints 11
func(1,2)        # Prints 15
func(1,2, d=5)   # Prints 17
func(a=1, 2)     # Error – can only name default params
```

  - All default params must come after non-defaults

RETURN VALUES

- A way to return a value from a function
  - Makes the function more usable.
- Return values we've already seen.

```
x = int(f)
print(3 + round(3.14, 1))
```

  - The int function *returns* a integer version of f.
  - This can be used as part of a larger expression.

RETURN VALUES, CONT.

- Example:

```
def adder(a, b):
    return a + b
x = adder(5, 7)   # x holds 12
print(adder("AB", "XY"))  # prints 'ABXY'
```

- If you don't include a return statement, python *silently* returns the value **None**.
  - e.g.

```
output = print("Test")
print(output)   # prints None
```

RETURN VALUES, CONT.

- Three types of return statements:

  `return`

  - returns None

  `return` *`expression`*

  - Evaluates the expression and returns it.
  - Can be any python object
    - int, float, list, tuple, pygame surface, etc.

RETURN VALUES, CONT.

- A return statement immediately ends a function call and returns to the caller

```
def func():
    print("You'll see me…")
    return
    print("…but you won't ever see me.")
```

- Often used to "back out" of a function (often in error cases)

```
def divide(a, b):
    if b == 0:   # Division would cause an error
            return 0.0   # Later, raise an exception
    return a / b # We won't get here if b is 0.
```

SCOPE AND GLOBAL VARIABLES

- **Scope** is a set of statements in which a name (usually of a variable) is valid.
- Python has two types of scope:
  - File (global) scope: your entire script
    - Any variable defined outside of a function is global.
  - Function (local, temporary) scope: the set of statements in a function.
    - Any variable defined inside a function has this scope.
    - This variable us un-defined after the function ends.
    - A local can have the same name as a global – it normally *masks* the global variable.

SCOPE AND GLOBAL VARIABLES

```
def foo(a, b):
     # a and b are local variables
     c = 9  # This creates another.
     print(a + b + c)

x = 99   # This is a global variable
#print(c)# Error! c only exists in foo.
a = 5     # This is a global a.  It is *totally* separate from
           # the local a above.
foo(x, a)   # Note, the value of a (5) is copied to the local b.
```

SCOPE AND GLOBAL VARIABLES.

- Using global variables in a function
  - First rule: **DON'T DO IT!**
    - Hurts readability, creates "spaghetti code", breaks encapsulation
  - Second rule: if you must do it, know how to use them.

```
def printGold():
        print("You have $" + str(gold))  # OK.  Uses the global
def addGold(amt):
        gold += amt     # Error – gold doesn't exist here (scope).
def loseGold():
        gold = 0        # Works, but creates a new LOCAL;
                        # doesn't change global gold.
def loseGold2():
        global gold
        gold = 0        # Works as expected now.

gold = 100      # Global variable
printGold()     # Prints $100
loseGold()
printGold()     # Still $100
loseGold2()
printGold()     # Now it's 0.
```

EXAMPLE OF A BAD USE OF GLOBALS

```
def drawFace():
        global x, y
        pygame.draw.circle(screen, (255,0,0), (x,y), 20, 0)
        # Other drawing code
        pygame.event.pump()
        pressed = pygame.key.get_pressed()
        if pressed[pygame.K_LEFT]:
                x -= 1
        # Other input commands

x = 400
y = 300
```

Problems:

- Style / Readability:
- Event "eating":
- Reusability:

EXAMPLE OF BAD GLOBAL USAGE, FIXED.

```
def drawFace(surf, x, y, color):
        pygame.draw.circle(surf, color, (x,y), 20, 0)
        # Other drawing commands

player1X = 100;   player1Y = 100
player2X = 700;   player2Y = 500

# Inside game loop
        pygame.event.pump()
        pressed = pygame.key.get_pressed()
        if pressed[pygame.K_LEFT]:   player1X -= 1
        # Other player 1 keys
        if pressed[pygame.K_w]:      player2X -= 1
        # Other player 2 keys

        # Drawing code
        drawFace(screen, player1X, player1Y, (255,0,0))
        drawFace(screen, player2X, player2Y, (0,0,255))
```

MAKING YOUR OWN MODULES

- Make a new file, **my_module.py**
  - Define a function **f** (2 integer args)
- Make a new file, **tester.py**

```
import my_module   # Notice missing .py


x = my_module.f(5, 7)
```

- Useful for:
  - making libraries of re-usable code.
  - Breaking up a large program

MUTABILITY

- Recall from the last lecture:
  - Strings and tuples are **immutable**
    - *So are "normal" variables (integers, floats, etc.)*
  - *Lists and Dictionaries are **mutable***
    - all objects are too
      - Pygame surfaces
      - sound objects
    - Later on, we'll make our own objects and classes – those are mutable too.

A SIMPLE (SURPRISING?) EXAMPLE

```
L = [1, 2, 3]
print(L)
G = L
print(G)
L[0] = 99
print(L)
print(G) – surprising?
```

- This happens because G and L refer to the same chunk of memory. The variables themselves are just **pointers**.
- So…variables that hold an immutable value hold the value itself. variables that hold a mutable value just hold a **reference** to it. This is good!

MUTABILITY AND FUNCTIONS

[Make a function that updates a list of something's]

RECURSION

- Simple definition: a function which calls itself
- Why?
  - An elegant solution to many problems
- [Example1: Factorial function]
- [Example2: Fibonacci non-recursively and then recursively]
- Important ideas:
  - The recursive function must have a **base case** (non-recursive)
  - Each recursive call must whittle the problem down so we eventually get to the base case.
    - Otherwise, you have something very similar to an infinite loop
    - Often you get a “stack overflow” error in response.
- [Maybe example 3: recursively subdividing a cube]