DATA TYPES & OPERATORS

```
5//3 = 1        # regular int divide
-2//-3 = 0      # if < 1 return 0
-1//3 = -1      # for(-) round up, return(-)
4//-3 = -2      # doesn't matter which side

6%3 = 0         # perfect factor, no remaind
5%3 = 2         # smallest factor under, diff
2%3 = 2         # return L if L < R
0%3 = 0         # 0 % anything = 0
-4%3 = 2        # go over (-6), return diff
4%-3 = -2       # return(-) if on rightsize
-4%-1 = -1      # works normally ,return(-)

2+3*4           # order of operations
5-4-3           # - associates left to right
4**3**2         # ** associates right to left
**always go L to R except for exponent

type('abc') -> <class 'str'>  # rturns class
isinstance('abc',str) -> True # rturns bool
type('abc') == str -> True    # rturns bool
```

VARIABLES & FUNCTIONS

statement - line of code that performs an action. cannot be used in operations

expression - data value or operation that evaluates to a value

function - procedure/sequence of statements

Built-in Functions

```
bool(0)          -> False # True for any other #
float(42)        -> 42.0 # converts to float
int(2.8)         -> 2 # converts to int
abs(-5)          -> 5 # absolute value
max(2,3)         -> 3 # maximum
min(2,3)         -> 2 # minimum
pow(2,3)         -> 8 # to the power of (2**3)
round(2.354,1)   -> 2.4 # rounds with given digits
```

import math Functions

```
math.factorial   # factorial
math.floor       # rounds down
math.ceil        # round up
math.pi          # 3.14…
math.e           # 2.71…
math.fsum        # precise floats
math.sqrt        # square root
math.sin         # sine
math.cos         # cosine
math.tan         # tangent
math.degrees     # converts radians
math.radians     # converts degrees
```

```
variables inside function are local
variables outside function are global

***functions may take functions as argument
   def h(n):
       return n+5
   def f(g, x):
       return 2*g(x)
   print f(h,3) # prints 16

round(0.5)          # builtin round rturns 0
round(1.5)          # returns 2 ???
roundHalfUp(0.5)    # returns 1 :)
roundHalfUp(1.5)    # returns 2 :)

***roundHalfUp(d) is a function that uses
   import decimal

def abs(n):
   return n if (n >= 0) else -n #use sparingly
```

LOOPS

For Loops

```
for x in range(start, end, step)
   # start is defaulted to 0 if only one
     value is given to range (lower bound)
   # end is exclusive (loop would stop at
     end -1) (upper bound)
   # step is backward if negative value

for i in value:
   # loops through elements of value, could
     be string or list
```

While Loops

```
while (conditional):
   #evaluate conditional each time
```

more on next column >>>>>>>>>>>>>>>>>>>>>>>>

```
while (value in a):    # use while loops
   a.remove(value)       when there is an
                         indeterminate # of
                         iterations

x = 0                  # don't forget value
while (x < len(list)):    needs to increment
   ...                    to avoid ∞ loop
   x += 1
```

Templates

```
def isPrime(n):              # isPrime
   if n < 2:
      return False
   for factor in range(2,n):
      if n % factor == 0:
        return False
   return True

def nthPrime(n):             # nthTemplate
   found = 0
   guess = 0
   while (found <= n):
      guess += 1
      if isPrime(guess):
         found += 1
   return guess

def bestTemplate(element):   # bestTemp
   best =
   current =
   while... or for...        # loop thrgh element
      current =              # get current
      if current > best:     # conditional
          best = current     # set best
      else:
          current =          # reset if needed
   return best               # return result
```

STRINGS

```
\"   # can add " to string
\\   # can add \ to string
\t   # adds tab
\n   # adds newline

for i in range(len(s)):  # loop w/ index
for c in s:              # loop w/out index

for name in names.split(','):
   # loops between commas
for line in text.splitlines():
   # loops between lines

strings do not alias like lists…
s = 'abc'
t = s          t = 'abc'
s += 'def      s = 'abcdef' but t = 'abc'

ord(character) # converts character to ascii number
chr(number)    # converts ascii number to character
eval(string)   #evaluates string if its a function
```

String Methods

```
s.isalnum()     # returns true if s is letters and numbers
s.isalpha()     # returns true if s is letters only
s.isdigit()     # returns true if s is numbers only
s.isnumeric()   # returns true if s is numbers only like ^^
s.islower()     # returns true if s is lowercase only
s.isspace()     # returns true if s space only
s.isupper()     # returns true if s is uppercase only
s.lower()       # converts string to all lowercase
s.upper()       # converts string to all uppercase
s.replace(elem, replacement, occurrences)
   # replaces element in str with replacement…however many times
s.count(elem)          # counts element
s.startswith(elem)     # boolean if string start w/ element
s.endswith(elem)       # boolean if string ends w/ element
s.find(elem)           # returns first index of element
                       # if none, returns -1?
s.index(elem)          # returns first index of element SIM^
```

Basic File IO?

GRAPHICS

```
from cmu_112_graphics import

def redrawAll(app, canvas):
   pass

runApp(width=400, height=400)

canvas.create_line(x0,y0,x1,y1)
   # line from x0,y0 to x1,y1
canvas.create_rectangle(x0,y0,x1,y1)
   # rectangle from x0,y0 to x1,y1
canvas.create_oval(x0,y0,x1,y1)
   # ellipse from corner x0,y0 to corner x1,y1
canvas.create_text(x0,y0,text="string")
   # text at x0,y0
```

```
attributes we can give to canvas.create:
   fill = 'color'
   outline = 'color'
   width = int                  # outline or line
   font = font size & BUI       # only for text

def drawClockFace(app, canvas):
   width = app.width
   height = app.height
   (cx,cy,r) = (width/2,height/2,min(with,height)/3)
   canvas.create_oval(cx-r,cy-r,cx+r,cy+r)
   r = 0.85
   for hour in range(12):
      hourAngle = math.pi/2 - (2*math.pi)*(hour/12)
      hourX = cx + r * math.cos(hourAngle)
      hourY = cy - r * math.sin(hourAngle)
      label = str(hour if (hour > 0) else 12)
      canvas.create_text(hourX, hourY, text=label,
                         font = "Ariel 16 Bold")
```

STYLE

Clarity Rules

```
Ownership            # 2pts no name/andrew ID
Comments             # 5pts no comments
                     # 2pts too many comments
Helper Functions     # 5pts no helper functions
                     # 2pts too many helper functions
                     # 2pts > 30 lines
Variable Names       # 5pts no meaningful variable names
                     # 2pts using some ^^
                     # 2pts not using camelCase
                     # 2pts using a built-in function name
Unused Code          # 2pts dead code
Formatting           # 2pts not using camelCase
```

Robustness Rules

```
Test Functions       # 5pts no test functions
                     # 2pts not enough
Efficiency           # 5pts more than 30 sec to run
                     # 2pts more than 5 sec to run
Repetitive Code      # 5pts 10 or more instances
                     # 2pts 3 or more instance
Magic Numbers        # 2pts using any magic numbers
If/Else              # 2pts not joining multiple ifs
Globals              # 2pts using any global variables
```

1D LISTS

List Functions

```
len(a) #returns length of string or list
min(a) #returns min value in list of ints
max(a) #returns max value in list of ints
sum(a) #returns sum of vals in list of ints

A = [1,2,3,4]
B = [1,2,3,4]

A == B               # returns True
A is B               # returns False

B = copy.copy(A)
A is B               # returns False

B = A
A is B               # returns True
```

is for aliases

```
ways to copy
   a = [2:3]
   b = copy.copy(a)
   c = a[:]
   d = a + [ ]
   e = list(a)

destructive modification - modifies the
   original value directly
non-destructive modification - creates new
   list and does not modify the original
```

List Methods

```
L.append()
   # adds element at the end of the list
L.clear()
   # removes all the elements from the list
L.count(value)
   # returns number of elements with the specified value
L.extend(value or elements)
   # adds the elements of a list (or any iterable), to the end
     of the current list
L.index(value, where to start looking)
   # returns the index of the first element with the specified
     value *crashes if value is not in list
L.insert(index, element)
   # adds an element at the specified position
L.pop(index)
   # removes the element at the specified position (last
     element is none specified) # returns value
L.remove(value or element)
   # removes the first item with specified value or element
L.reverse()
   # reverses order of list
L.sort()
   # sorts the list from lowest to highest
```

more on other side >>>>>>>>>>>>>>>>>>>>>>>

slicing lists

```
a = [2,3,5,6]
b = a[:2] + [4] + a[2:]
print(a)                    [2,3,5,6]
print(b)                    [2,3,4,5,6]
```

Looping over Lists

```
for index in range(len(L)):    # loops through index
   L[index]...

for item in L:                 #loops through item
   item...

index = 0                      #increment index
while (index < len(L)):
   ...
   index += 1

while item in L:               #checks for item
   L.remove(item)
```

Tuples

```
t = (1,2,3)     # tuples are immutable! They cannot change

g = [1,2,3]     # tuple from list
t = tuple(g)

a, b, c = t     # can do parallel assignment with tuples
a = 1
b = 2
c = 3
```

list comprehensions

```
L = [i for i in range(10)]     is the same as...

L = []
for i in range(10):
   L.append(i)          both are [0,1,2,3,4,5,6,7,8,9]
```

Converting Between Lists & Strings

```
a = list("string")
a = ['s','t','r','i','n','g']

a = "How are you today?".split(" ")
a = ['How', 'are', 'you', 'today?']

b = "XX".join(a)     #works opposite of .split()
b = 'HowXXareXXyouXXtoday?'
```

2D LISTS

```
rows = 2
cols = 3
a = [([0] * cols) for row in range(rows)]

a = [[0,0,0],
     [0,0,0]]              # list comprehension ^^

a[0][0] = 5                # list index assignment
a = [[5,0,0],
     [0,0,0]]

rows, cols = len(a), len(a[0])
   #gets list dimensions as parallel assignment

e = [[1,2,3],
     [4,5,6]]

f = copy.copy(e)           # copy.copy() creates
f = [e[0],e[1]]              shallow copy

g = copy.deepcopy(e)       # copy.deepcopy()
g = [[1,2,3],                creates deep copy
     [4,5,6]]              # still has limits tho
```

Nested Loops over 2D Lists

```
for row in range(rows):        # col index by rows
   for col in range(cols):
      ...
      ...
      a[row][col]...

for col in range(cols):        # row index by cols
   for row in range(rows):
      ...
      ...
      a[row][col]

rowList = L[row]               # access whole row

colList = []                   # access whole col
for row in L:
   colList.append(row[col])

colList = [row[col] for row in L]
   # list comprehension
```

more on next column >>>>>>>>>>>>>>>>>>>>>>>

Word Search

```
def wordSearch(board, word):
    rows, cols = len(board), len(board[0])
    for row in range(rows):
        for col in range(cols):
            result = wordSearchFromCell(board, word, row, col)
            if (result != None):
                return result
    return None

def wordSearchFromCell(board, word, startRow, startCol):
    for drow in [-1, 0, 1]:
        for dcol in [-1, 0, 1]:
            if (drow, dcol) != 0:
                result = wordSearchFromCellInDirection(board, word,
                                        startRow, startCol, drow, dcol)
                if (result != None):
                    return result
    return None

def wordSearchFromCellInDirection(board, word, startRow, startCol,
                                                        drow, dcol):
    dirNames = [["up-left"  ,  "up" ,   "up-right"],
                ["left"     ,  ""   ,      "right"],
                ["down-left", "down", "down-right"]]
    for i in range(len(word)):
        row = startRow + i*drow
        col = startCol + i*dcol
        if ((row < 0) or (row >= rows) or
            (col < 0) or (col >= cols) or
            (board[row][col] != word[i])):
            return None
    return (word, (startRow, startCol), dirNames[drow+1][dcol+1])
```

**review Connect4

ANIMATIONS 1

```
from cmu_112_graphics import

def appStarted(app):                # app is started
   app.variables                    # stores variables

def keyPressed(app, event):         # changes variables
   app.variable...changed, or
   controllerFunction()

def redrawAll(app, canvas):         # draws variables
   canvas.create_...or
   drawFunction()
```

Model-View-Controller

```
model: contains all the data for the
   animation. stores the model in app
   object's attributes
view: draws the app using the values in the
   model
controller: responds to keyboard, mouse,
   timer, & other events...updates model

controller<-------->model--------------->view
controller<-----------//---------------->view

def keyPressed(app, event):          # moves app.cx
   if (event.key == 'Left'):           left & right
      app.cx -= 10
   if (event.key == 'Right'):
      app.cx += 10

def keyPressed(app, event):          # moves app.cx
   if (event.key == 'Left'):           with bounds
      app.cx -= 10
      if (app.cx - app.r < 0):
          app.cx = app.r
   if (event.key == 'Right'):
      app.cx += 10
      if (app.cx + app.r > app.width):
          app.cx = app.width - app.r

def keyPressed(app, event):          # moves app.cx
   if (event.key == 'Left'):           w/ wraparound
      app.cx -= 10
      if (app.cx - app.r < 0):
          app.cx  = app.width + app.r
   if (event.key == 'Right'):
      app.cx += 10
      if (app.cx + app.r > app.width):
          app.cx = 0 - app.r

def keyPressed(app, event): # moves app.cx 2 dimens
   if (event.key == 'Left'):    app.cx -= 10
   elif (event.key == 'Right'): app.cx += 10
   elif (event.key == 'Up'):    app.cy -= 10
   elif (event.key == 'Down'):  app.cy += 10

def mousePressed(app, event):
   app.cx = event.x             # moves w/ mouse press
   app.cy = event.y

def timerFired(app):            # moves app.cx w/ timer
   app.cx -= 10
   if (app.cx - app.r < 0):
          app.cx  = app.width + app.r

** review pausing with a timer
   # timerFired(app) -> if not paused: doStep
   # doStep = tireFired above^^
   # keyPressed ->'p': app.paused,'s': doStep
```

OBJECT-ORIENTED PROGRAMMING

```
classes are also called "types" in python
   ex: int, float, str, bool
instances are values of a given clss or type
   ex: 'abc' is a string instance
```

Namespaces

```
from types import SimpleNamespace

dog1 = SimpleNamespace(name='rex',age=4,breed='mutt')
print(dog1) #namespace(age=10, breed='mutt', name='rex')
print(dog1.name) # rex

dog1.name = 'sparky'
print(dog1) # namespace(age=4, breed='mutt', name='sparky')
print(dog1.name) #s parky

dog2 = SimpleNamespace(name='max',age=14,breed='shp')
dog3 =
   SimpleNamespace(name='sparky',age=4,breed='mutt')

dog1 == dog2 # False
dog1 == dog3 # True
```

Dataclasses

```
a dataclass is like a SimpleNamespace but better:
   -it has required fields
   -it has custom types

from dataclasses import make_dataclass

Dog = make_dataclass('Dog',['name','age','breed'])
dog1 = Dog(name='rex', age=4, breed='mutt')

print(dog1) # Dog(name='rex', age=4, breed='mutt')
print(dog1.name) # rex

dog1.name = 'sparky'
print(dog1) # Dog(name='sparky', age=4, breed='mutt')
print(dog1.name) # sparky

dog2 = Dog(name='max', age=14, breed='shp')
dog3 = Dog(name='sparky', age=4, breed='mutt')

dog1 == dog2            # False
dog1 == dog3            # True

type(dog1)              #  <class 'types.Dog'>
isinstance(dog1, Dog)   # True
```

ANIMATIONS 2 & 3

```
def getCell(app, x, y):
   if (not pointInGrid(app, x, y)):
      return (-1, -1) # value specific to notes
   gridWidth = app.width - 2*app.margin
   gridHeight = app.height - 2*app.margin
   cellWidth = gridWidth / app.cols
   cellHeight = gridHeight / app.rows
   row = int((y - app.margin) / cellHeight)
   col = int((x - app.margin) / cellWidth)
   return (row, col)

def getCellBounds(app, row, col):
   gridWidth = app.width - 2*app.margin
   gridHeight = app.height - 2*app.margin
   cellWidth = gridWidth / app.cols
   cellHeight = gridHeight / app.rows
   x0 = app.margin + col*cellWidth
   x1 = x0 + cellWidth
   y0 = app.margin + col*cellHeight
   y1 = y0 + cellHeight
   return (x0, y0, x1, y1)

def pointInGrid(app, x, y):
   return ((app.margin<=x<=app.width-app.margin) and
           (app.margin<=y<=app.height-app.margin))
```