# Welcome to ENGR102

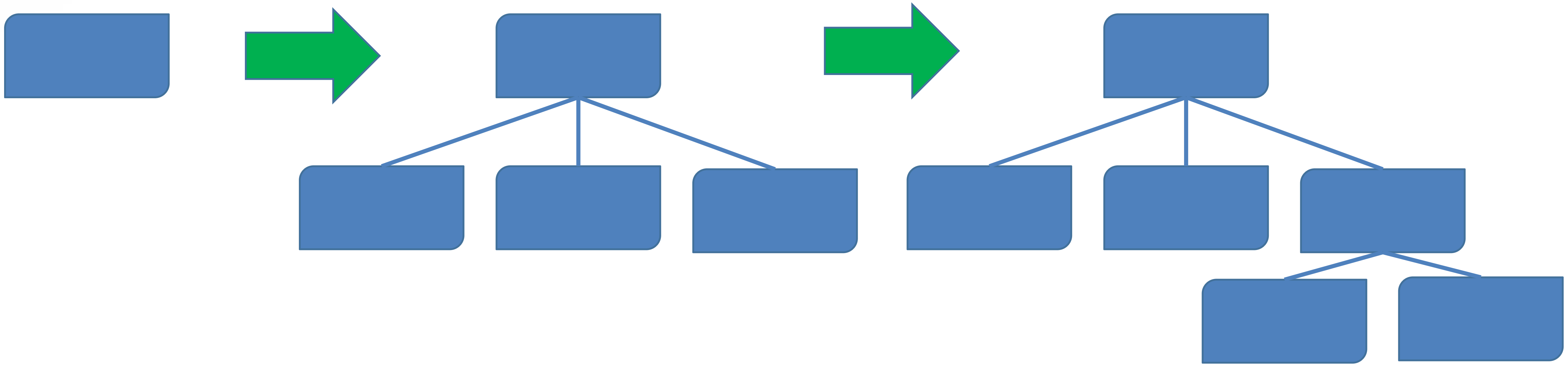
Cable Kurwitz

## What are we going to cover today?

- Top-down design of functions
- Bottom-up design
- Docstrings and documenting functions

## Reminder: Top-Down Design

- Take a complex task and break it into more manageable pieces
- Repeat the process until the individual parts are basic enough to be "obvious"

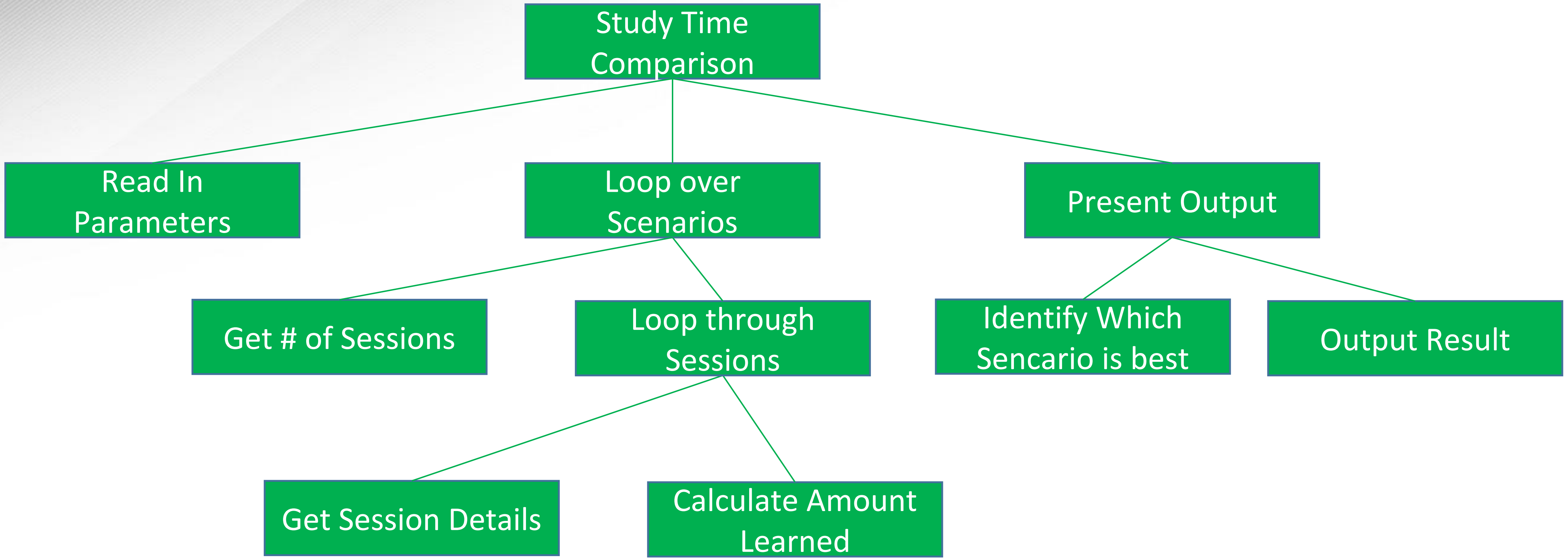


## Top-Down Design With Functions

- Essentially, each "block" can be thought of as a function.
    - A coherent unit of actions, tied together
- The smallest block is a single function that doesn't call other functions.
- The upper levels of the hierarchy might call multiple functions.
- The end result is that any one function should be easy to understand on its own.
    - Doesn't need to worry much about lower levels in the hierarchy
    - Doesn't need to worry much about higher levels in the hierarcy

## Example

- Imagine we have a program where we want to evaluate different models of studying. We want to determine which among several different options will help us learn the most.
- Basic Outline
  - Get the parameters governing how effective study is
  - Loop to get scenarios
  - Output results
- Then break down from there


Study Time Comparison
Read In Parameters
Loop over Scenarios
Present Output
Get # of Sessions
Loop through Sessions
Identify Which Sencario is best
Output Result
Get Session Details
Calculate Amount Learned

## Converting to Code

- Each of the blocks will become its own function:

```
def getsessiondetails():


def computelearned():


def findbestscenario():


def outputbestresult():


def getsessions():
```

## Converting to Code

- The main program will call the first level functions, in order
- The non-leaf functions will call their children functions

- Notice the order of function calls
  - The children functions are listed before the parent functions, so that the parent functions can call them
- Notice that we don’t have any variables or parameters yet!

```
def getsessiondetails():

def computelearned():

def findbestscenario():

def outputbestresult():

def getsessions():

def loopsessions():
    getsessiondetails()
    computelearned()
```

```
def readparams():

def loopscenarios():
    getsessions()
    loopsessions()

def presentoutput():
    findbestscenario()
    outputbestresult()

readparams()
loopscenarios()
presentoutput()
```

## Filling in Code

- Next, you want to determine what parameters need to be passed in, and what values returned by functions.
- For example, the “read parameters” function does not need to take any parameters, and returns the information needed to compute study session effectiveness:
    - Starting learning rate
    - Warmup time
    - Steady learning rate
    - Fatigue time
    - Time at which no more learning occurs
- We’d want to read in this data (e.g. from a user via the console) and return it from the function as a tuple.

```
def getsessiondetails():
    return

def computelearned():
    return

def findbestscenario():
    return

def outputbestresult():
    return

def getsessions():
    return

def loopsessions():
    getsessiondetails()
    computelearned()
```

```
def readparams():
    initrate = float(input("What is the initial rate of concepts learned/minute? "))
    warmuptime = float(input("How long will it take you to warm up? "))
    plateaurate = float(input("What is the rate of concepts learned/minute once warmed up? "))
    fatiguetime = float(input("How long will it take to get fatigued, from the time you start? "))
    stoptime = float(input("What is the point at which you are no longer learning anything? "))
    return (initrate, plateaurate, warmuptime, fatiguetime, stoptime)
def loopscenarios():
    getsessions()
    loopsessions()
def presentoutput():
    findbestscenario()
    outputbestresult()
(rate_start, rate_steady, t1, t2, t3) = readparams()
loopscenarios()
presentoutput()
```

## Filling in code

- Likewise, we should be able to fill in the other portions of the code
  - But will skip the details here for time purposes.

## Advantages to different functions

- By dividing the code up in this way, we make it easy to make changes.

- For example, say we wanted to read the study parameters from a file, instead.
  - We would just change the implementation of the “getparameters” function, and possibly the call to that function

- We have separated the specifics of **how** this is done from the **goal** of the routine!
  - The calling routine does not have to worry about how the function underneath works
  - The function can be implemented without worrying about how its parameters get used

## Two options

```
def readparamsA():
    initrate = float(input("What is the initial rate of concepts learned/minute? "))
    warmuptime = float(input("How long will it take you to warm up? "))
    plateaurate = float(input("What is the rate of concepts learned/minute once warmed up? "))
    fatiguetime = float(input("How long will it take to get fatigued, from the time you start? "))
    stoptime = float(input("What is the point at which you are no longer learning anything? "))
    return (initrate, plateaurate, warmuptime, fatiguetime, stoptime)
(rate_start, rate_steady, t1, t2, t3) = readparamsA()
def readparamsB():
    infile = open("StudyParams.dat",'r')
    initrate = float(infile.readline())
    warmuptime = float(infile.readline())
    plateaurate = float(infile.readline())
    fatiguetime = float(infile.readline())
    stoptime = float(infile.readline())
    return (initrate, plateaurate, warmuptime, fatiguetime, stoptime)
(rate_start, rate_steady, t1, t2, t3) = readparamsB()
```

# Top-down design with functions

- Top-down design usually stops when the implementation is "obvious"
- It doesn't always make sense to create a separate function
  - e.g. if a task takes 1 line, then why call a function?  Just write 1 line of code.
- Generally, your functions should not be "too large"
  - They should have a clear purpose/goal
  - They should be small enough that someone can read them and easily understand everything they are doing
  - If you find your function is doing very different things, or is too large, then split it into different functions.
    - Either call a function from inside this function (create children)
    - Or, divide the function into two separate calls (split a node into two)

## An alternative design approach

- Top-down decomposition of a problem is not the only approach
  - Though it is one of the best, and usually can be tried first
- Another approach that works well with functions is the bottom-up approach.

## Bottom-Up Design

- In programming, bottom-up design refers to taking existing, simpler pieces of code and combining them to create more complex pieces of code.
- As you think about a problem, think about the things that you are likely to need to do, based on what you already know
- When you identify a basic task that you can code up in a short function, go ahead and do so.
- As you build more functions, the things you can build easily become more and more complicated
  - Eventually, you should be able to build your “goal” program directly

## Analogy: planning a vacation

- Earlier, we talked about what a top-down vacation plan would look like. What might a bottom-up process look like?
- Say you want to take a trip to New York City.
  - You might start by planning activities you might want to do. For example:
    - Outline what a visit to the Statue of Liberty would look like; or the 9/11 memorial; or the Empire State Building, etc.
      - How do you get there, what do you do while there, how long to spend, etc.
    - Or, look at what a visit to one of the museums would involve
      - Arrival, plan for visiting exhibits, time spent, etc.
    - Look into how you would spend an evening seeing a Broadway show – meal, show, after show plans
  - Then, start combining those:
    - E.g. morning at a museum, evening at a Broadway show
  - Eventually you'll put together a plan for a whole trip
    - But, you might not use everything – maybe you have to skip a museum, etc.
    - And, you might find you have to fit some things in later – e.g. did you plan transportation?

## When to Create a new function

- When you identify something that will be done repeatedly
  - e.g. you might expect that you'll need to convert Fahrenheit to Celsius repeatedly.  So, create a function that will do that for you.
- When you identify something that's a key concept, that can be built easily from existing code/functions
  - e.g. you know that your program will need to compute statistics from lists of data, so write a routine that will do that for you

## Example – writing with turtle graphics

- Turtle graphics are a very simple “drawing” routine
  - Used more for learning/playing than for actual production, but it is easy to illustrate ideas
  - The turtle module includes turtle graphics commands
- The “turtle” is something that has a few very basic commands:
  - Move forward/backward
  - Turn left/right
  - Put pen down/up
  - And some others, but that’s enough to illustrate…
- As it moves, it “draws” a path

## Drawing a square

```
from turtle import *
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)  #To return to starting orientation
input()  #So that the window doesn't disappear
```

## Creating a function

```
from turtle import *

def drawsquare():
    forward(100)
    left(90)
    forward(100)
    left(90)
    forward(100)
    left(90)
    forward(100)
    left(90)  #To return to starting orientation

drawsquare()
input()  #So that the window doesn't disappear
```

## Functions

- Now we have a new turtle “function” – drawing a square.
  - Any time we want to draw a square from the current position, we just call that routine
- For example, we could draw a square, move, and draw another square

```
from turtle import *
def drawsquare():
    forward(100)
    left(90)
    forward(100)
    left(90)
    forward(100)
    left(90)
    forward(100)
    left(90)  #To return to starting orientation

drawsquare()
up()  #Pick up pen - no drawing
forward(150)
down()   #Put pen down and start drawing
drawsquare()

input()  #So that the window doesn't disappear
```

## Example

- If we wanted to use turtle graphics to write out a name, how might we do it?
    - Create functions for writing each letter and moving over
    - Create a function to take a single letter and call the right function
    - Create a function to take in a string name and write letter by letter
- We could build this bottom-up

## Turtle graphics and robotics

- Turtle motion is very similar in behavior to many robotic controls
  - Very simple controls: actuate particular motors
- Just like we combine turtle motions to make more complex shapes, can combine robot motions to create more complex actions
  - Move motors to cause a robot to move forward by an increment, or turn, or grasp, etc.
  - Then, combine these to make more complex motions

# Bottom-Up Programs

- We still have a hierarchy of functions
  - Higher level functions call lower level functions
  - But, the lower-level functions are designed first
- Sometimes top-down and bottom-up will yield similar designs, but not always
- Bottom-up approaches tend to be used to develop modules
  - More complex routines, built on top of more basic ones
  - Can be used in other programs to perform more complex tasks

# Top-Down vs. Bottom-Up

**Top-Down Design**

- Creates a hierarchy of functions
- All code is clearly directed toward the goal
    - Every function has its place
- Code can become too specialized, missing chances to re-use ideas
- While developing, much code is in an incomplete state

**Bottom-Up Design**

- Creates a hierarchy of functions
- Can promote code re-use
    - common concepts are identified
- Code can be tested more easily
    - Every time a function is finished
- Code development tends to be less-focused on final goal

## Design in Practice

- Top-Down and Bottom-Up are not the only design approaches
  - Object-oriented design, which is very common now, takes a different approach, combining aspects of both top-down and bottom-up, with some other design approaches (encapsulation, inheritance, polymorphism)
- People seldom use a strict top-down or bottom-up approach
  - Usually, aspects of each are used, depending on the problem and the goals
  - One example:
    - Start building bottom-up, thinking about functionality you might need
    - Build routines that seem like they'd be commonly called
    - Once you have a richer "base" of code, work top-down
    - You won't have to go as far in the top-down approach, since you have more useful routines available to you

## Docstrings

- As we write functions, it becomes important to document them
- Functions are one of the most important things to document
  - What is their purpose?
  - What input do they need (parameters)?
  - What do they return?
- There is a special way for writing comments about functions
  - This also makes it possible to learn about functions, e.g. in an interactive Python routine, or to print out information about a function in a program.
  - Called a “docstring”, it is used to document functions in Python
- Can access it by using the “help” command:
  - `help(<function_name>)`

## Docstrings

- After the function definition, the first line of code should be a string
  - Designated with a triple-quote string
  - Can take multiple lines if needed

- It is good practice to create a docstring for each of your functions
  - State what it does.  Then give information about parameters needed, default parameter values, return values, etc.

## Example

```
from turtle import *

def drawsquare():
    '''Draw a square and return to original position and orientation.'''
    forward(100)
    #etc. (remaining code removed here for space reasons)

help(drawsquare)
```

**Console**

```
Help on function drawsquare in module __main__:
drawsquare()
    Draw a square and return to original position and orientation.
```

## A Final Summary: Abstraction

- The key to why top-down design, bottom-up design, and functions are useful is that we can ignore the details of how other parts of the program work, while we focus on one particular part.
- This is called **abstraction**, and it's one of the most important concepts in computing!
    - This goes beyond just software design.
    - Many aspects of computing – from the way we think about memory, to the way we think about a computer operating, to the way we organize data, to the way we design algorithms – all involve creating abstractions.
    - It also has application outside of computing, to any complex task we face.

# Abstraction When Designing Functions

- A function should not have to worry about the thing calling it
  - It just does its job with whatever parameters are passed in
- A function should not have to worry about the things it calls
  - They should just do their job based on what parameters are passed in
- It is very important, especially in larger software projects, to manage complexity.
  - Creating and using functions like this is one of the key ways to manage complexity
  - There are a variety of other techniques, also, that you'll encounter in more advanced programming