

Algorithmic Thinking: Loops and Conditionals



Announcements

- Programming Assignment 2 due tonight at 11:59 via Autolab

- Tomorrow:
 - Problem Set 3
 - Lab 3
 - Programming Assignment 3
 - Note: updated test file

Today

- ▣ A control flow structure:
 - ▣ `for` loop
 - ▣ `while` loop
- ▣ `Range`
- ▣ Nesting control structures

- ▣ The notion of an algorithm

- ▣ Moving from algorithm to code

- ▣ Python control structures:
 - ▣ Conditionals

- ▣ `Lists` (?)

Iteration with **for loops**

```
def test1():  
    for i in range(1,6):  
        print("Woof")
```

```
>>> test1()  
Woof  
Woof  
Woof  
Woof  
Woof
```

What determines how many times “Woof” is printed is **the number of elements in the range.**

Any expression that gives 5 elements in the range would give the same output.

For example, `range(5)`, `range(0,5)`, ...

Iteration with `for` loops

```
def test2():  
    for i in range(1,6):  
        print(i)
```

```
>>> test2()  
1  
2  
3  
4  
5
```

`range(5)` ?

`range(0, 5)` ?

`range(1, 10, 2)` ?

`range(2, 10, 2)` ?

`range(10, 1, -1)` ?

Iteration with **for** loops

```
def test3():  
    for i in range(1,6):  
        print("Woof" * i)
```



This expression creates a string that concatenates *i* number of "Woof"s.

```
>>> test3()  
Woof  
WoofWoof  
WoofWoofWoof  
WoofWoofWoofWoof  
WoofWoofWoofWoofWoof
```

Analogy:

$3 * 4$ is equivalent to $4+4+4$

$3 * \text{"a"}$ is equivalent to
 $\text{"a"} + \text{"a"} + \text{"a"}$

Nesting?

An epidemic

Each newly infected person infects 2 people the next day.
The function returns the number of sick people after n days.

```
def compute_sick(d):  
    # computes total sick after d days  
    newly_sick = 1 # initially 1 sick person  
    total_sick = 1  
  
    for day in range(2, d + 1):  
        # each iteration represents one day  
  
        newly_sick = newly_sick * 2  
        total_sick = total_sick + newly_sick  
  
    return total_sick
```


Variation on the Epidemic Example

Let's write a function that

- ▣ **Inputs the size of the population**
- ▣ **Outputs the number of days left before all the population dies out**

How can we do that using iteration (loops)?

Keep track of the number of sick people.

But do we know how many times we should loop?

Recall the Epidemic Example

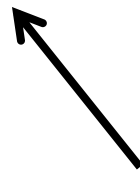
```
def days_left(population):  
    # computes the number of days until extinction  
    days = 1  
    newly_sick = 1  
    total_sick = 1  
    while total_sick < population:  
        # each iteration represents one day  
        newly_sick = newly_sick * 2  
        total_sick = total_sick + newly_sick  
        days = days + 1  
    print(days, " days for the population to die off")  
    return days
```

while loop

Format:

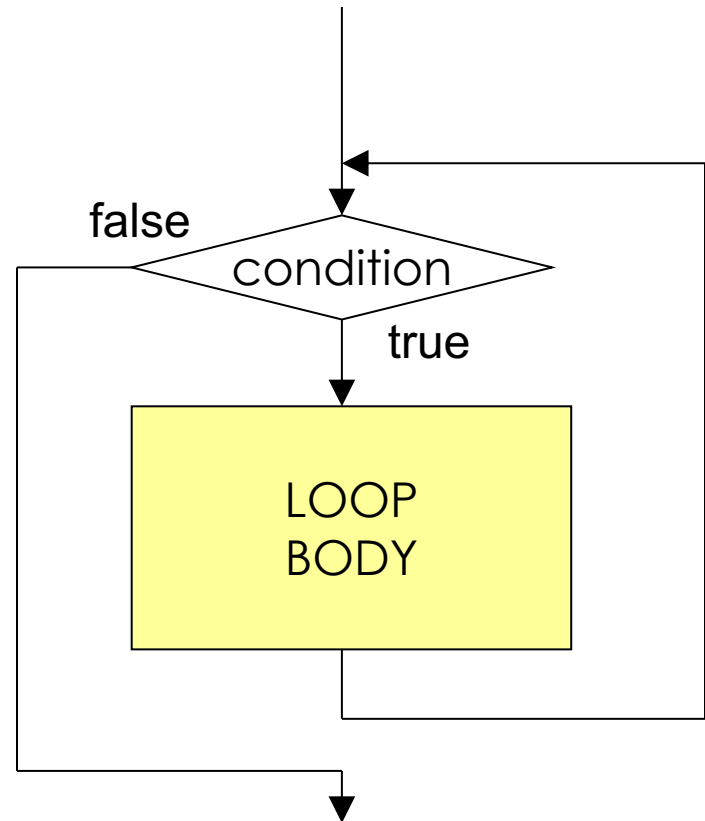
while *condition*:

loop body



one or more instructions
to be repeated


If the loop condition becomes false during the loop body, the loop body still runs to completion before we exit the loop and go on with the next step.



Recall the Epidemic Example

```
def days_left(population):  
    # computes the number of days until extinction  
    days = 1  
    newly_sick = 1  
    total_sick = 1  
    while total_sick < population:  
        #each iteration represents one day  
        newly_sick = newly_sick * 2  
        total_sick = total_sick + newly_sick  
        days = days + 1  
    print(days, "days for the population to die off")  
    return days
```

Loop condition



While Loop Examples

Prints first 10 positive integers

```
i = 1
while i < 11:
    print(i)
    i = i + 1
```

How about the following?

```
i = 0
while i < 10:
    i = i + 1
    print(i)
```

What is the value of *i* when we exit the loop?

While vs. For Loops

Prints first 10 positive integers

```
i = 1
while i < 11:
    print(i)
    i = i + 1
```

Prints first 10 positive integers

```
for i in range(1,11):
    print(i)
```

When to use `for` or `while` loops

- If you know in advance **how many times** you want to run a loop use a **`for`** loop.
- When you **don't know the number** of repetition needed, use a **`while`** loop.

Algorithms

- An algorithm is “a precise rule (or set of rules) specifying how to solve some problem.”
(thefreedictionary.com)
- The study of algorithms is one of the foundations of computer science.



Mohammed al-Khwarizmi (äl-khōwārēz´mē)

Persian mathematician of the court of Mamun in Baghdad...the word **algorithm** is said to have been derived from his name. Much of the mathematical knowledge of medieval Europe was derived from Latin translations of his works. (encyclopedia.com)

An algorithm is like a function

$$F(x) \rightarrow y$$



Input

- **Input specification**
 - Recipes: ingredients, cooking utensils, ...
 - Knitting: size of garment, length of yarn, needles ...
 - Tax Code: wages, interest, tax withheld, ...
- Input specification for computational algorithms:
 - **What kind of data** is required?
 - **In what form** will this data be received by the algorithm?

Computation

- An algorithm requires **clear and precisely stated steps** that express how to perform the operations to yield the desired results.
- Algorithms assume a basic set of ***primitive operations*** that are assumed to be understood by the executor of the algorithm.
 - Recipes: beat, stir, blend, bake, ...
 - Knitting: casting on, slip loop, draw yarn through, ...
 - Tax code: deduct, look up, check box, ...
 - Computational: add, set, modulo, output, ...

Output

- Output specification
 - Recipes: number of servings, how to serve
 - Knitting: final garment shape
 - Tax Code: tax due or tax refund, where to pay
- Output specification for computational algorithms:
 - **What results** are required?
 - **How** should these results be **reported**?
 - **What happens if no results can be computed** due to an error in the input? What do we output to indicate this?

Is this a “good” algorithm?

□ Input: slices of bread, jar of peanut butter, jar of jam

1. Pick up some bread.
2. Put peanut butter on the bread.
3. Pick up some more bread.
4. Open the jar of jam.
5. Spread the jam on the bread.
6. Put the bread together to make your sandwich.

□ Output?

What makes a “good” algorithm?

- A good algorithm should produce the **correct outputs for any set of legal inputs.**
- A good algorithm should **execute efficiently with the fewest number of steps as possible** and should **always stop.**
- A good algorithm should be designed in such a way that **others will be able to understand it and modify it to specify solutions to additional problems.**

A Simple Algorithm

Input numerical *score* between 0 and 100 and
Output “Pass” or “Fail”

Algorithm:

1. **If** *score* \geq 60
 - a. Set *grade* to “Pass”
 - b. Print “Pass”
2. **Otherwise**,
 - a. Set *grade* to “Fail”
 - b. Print “Fail”
3. Print “See you in class”
4. Return *grade*

Exactly **one** of step 1 or step 2 is executed, but step 3 and step 4 are **always** executed.

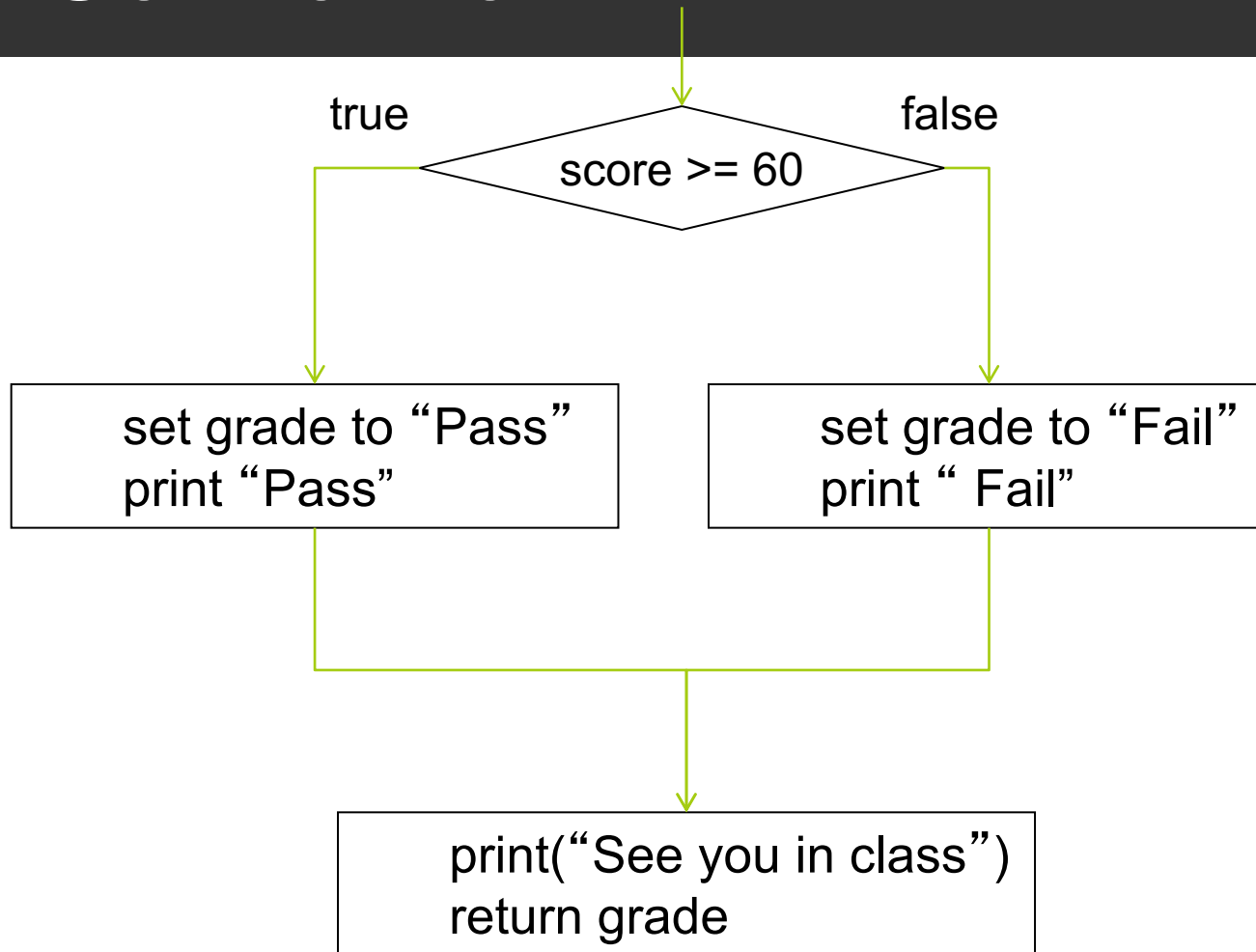
Coding the Grader in Python

Algorithm:

1. If *score* ≥ 60
 - a. Set *grade* to "Pass"
 - b. Print "Pass"
2. Otherwise,
 - a. Set *grade* to "Fail"
 - b. Print "Fai"
3. Print "See you in class "
4. Return *grade*

```
def grader(score):  
    if score >= 60:  
        grade = "Pass"  
        print("Pass")  
    else:  
        grade = "Fail"  
        print("Fail")  
    print("See you in class")  
    return grade
```

Control Flow

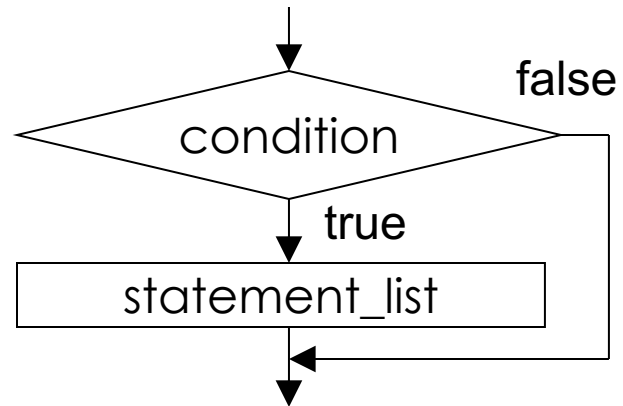


Flow chart: **if** statement

Format:

if *condition* :

statement_list



Flow chart: **if/else** statement

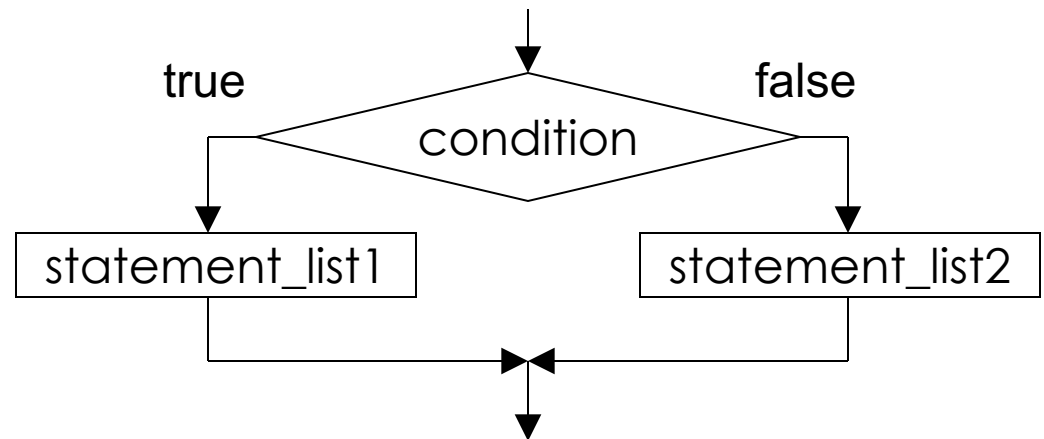
Format:

if *condition* :

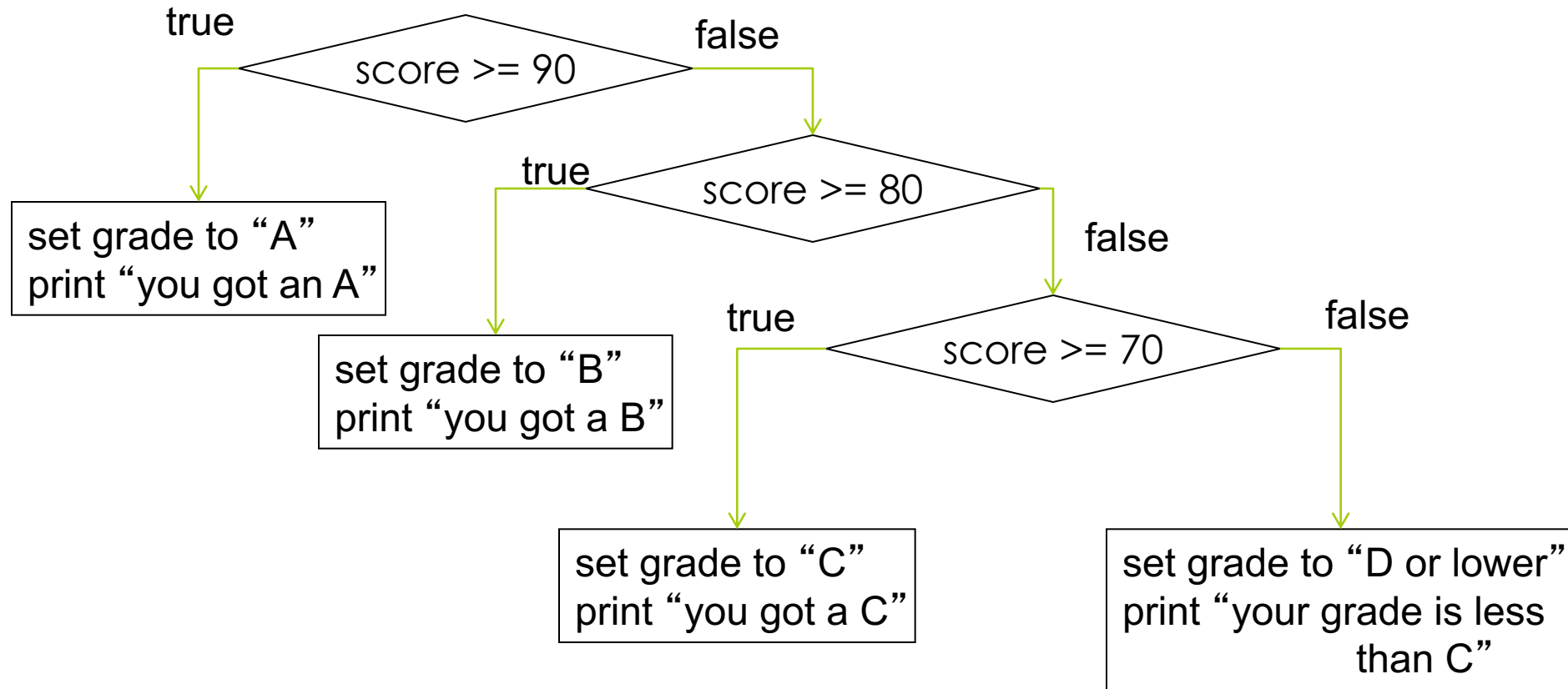
statement_list1

else:

statement_list2



Grader for Letter Grades



Nested if statements

```
def grader2(score):  
    if score >= 90:  
        grade = "A"  
        print("You got an A")  
    else: # score less than 90  
        if score >= 80:  
            grade = "B"  
            print("You got a B")  
        else: # score less than 80  
            if score >= 70:  
                grade = "C"  
                print("You got a C")  
            else: #score less than 70  
                grade = "D or lower"  
                print("Your grade is less than C")  
    return grade
```

Equivalently

```
def grader3(score):  
    if score >= 90:  
        grade = "A"  
        print("You got an A")  
    elif score >= 80:  
        grade = "B"  
        print("You got a B")  
    elif score >= 70:  
        grade = "C"  
        print("You got a C")  
    else:  
        grade = "D or lower"  
        print("Your grade is less than C")  
    return grade
```

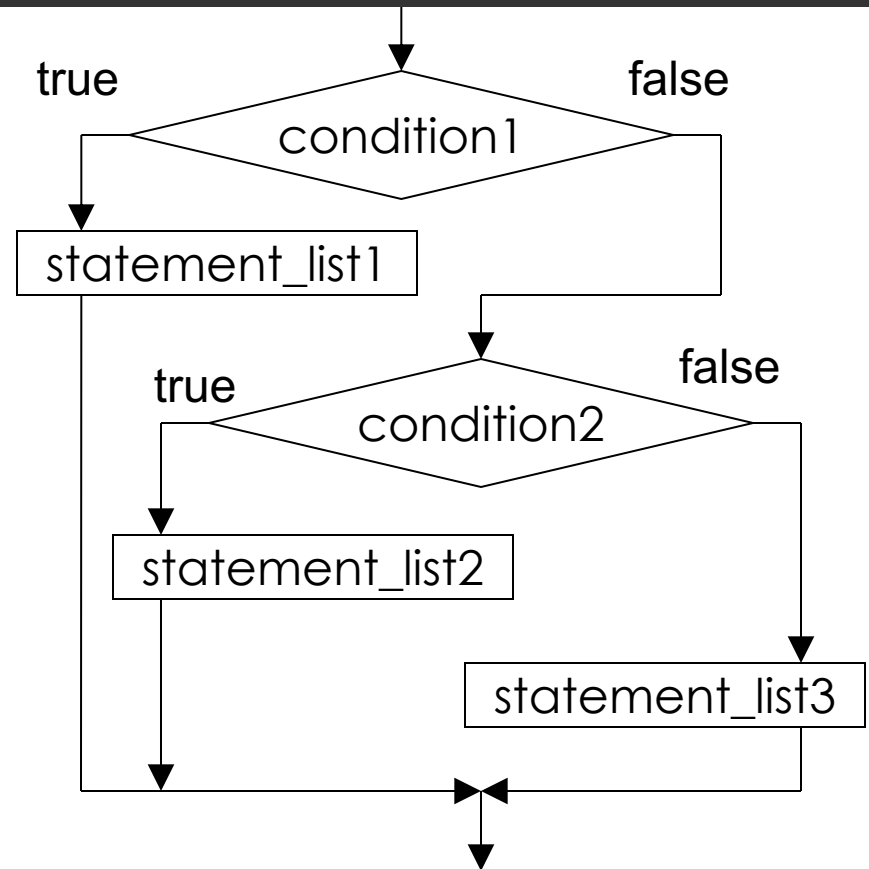
Flow chart: if/elif/else statement

Format:

```
if condition1 :  
    statement_list1
```

```
elif condition2 :  
    statement_list2
```

```
else:  
    statement_list3
```



Summary

- Notion of an algorithm:
 - Kinds of instructions needed to express algorithms
 - What makes an algorithm a good one
- Instructions for specifying control flow (for loop, while loop, if/then/else)
 - Flow charts to express control flow in a language-independent way
 - Coding these control flow structures in Python

Exercise

Write a function that returns how many of the three integers `n1`, `n2`, and `n3` are odd:

```
def num_odd(n1, n2, n3):
```

Exercise

Write a function that prints whether `die1` and `die2` are doubles, cat's eyes (two 1's) or neither of these.

```
def print_doubles(die1, die2):
```

Try:

- Saving money to buy a new car – how long will it take to save for a new Tesla Model X @ \$80,000. (5000.00 in a savings account)
- Saving for retirement – for different retirement targets, and calculate how long it will take to reach that target. Identify your variables and pre-assign values.
- Can you generalize the above to accommodate different user input?