Python Programming For Beginners

Module 4

Go over homework.

Show 2 different ways to code the homework.

1) Three separate calculations (2 variations)

2) One calculation

Guess The Number Program demo

In this module, we're going to talk about "loops". To demonstrate what this is, we'll build some simple game programs. By the end of this module, we'll build a "Guess the Number" program.

[Demo program running first.]

++++

- program chooses random target number between 1 and 20
- asks for user input
- compares users input to target
- gives feedback to user about higher, lower, or correct
- also it keeps track of how many guesses the user has made
- allows the user to make up to to 5 guesses

You already know how to do much of this. For example, you know how to get the user's guess (input statement).

What kind of statement might we use to do the comparison between the user's answer and our target number??

You see that we are doing some counting, and you already know how to take a variable and add one to it.

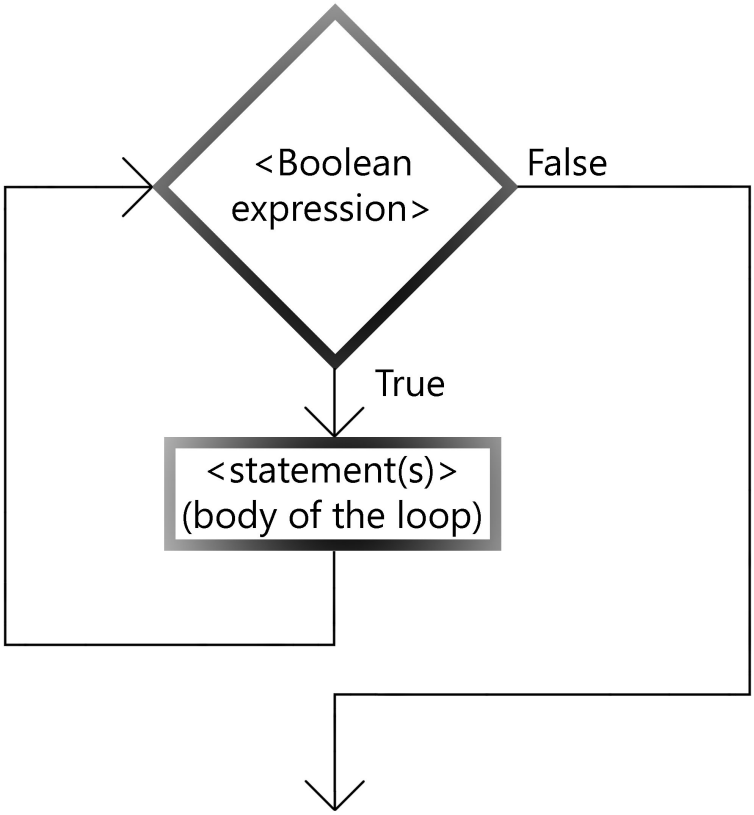
But, in order to build this full program, we need to learn a few more things about programming, and how to implement these things in Python.

Loops

When I ran the program, you probably noticed that it allowed the user to make multiple guesses. In order to do something like this, I need to introduce a new concept called a "loop".

+++++
DEF Loop: a block of code that is repeated until a certain condition is met.

+++++
Here is a flowchart of loop:



++++ Silly real life examples:

I am hungry
while hungry
    take a bite of food
    chew
    swallow
    if thirsty:
        take a drink

eat dessert!

++++++

ask son to take out garbage
anger = 0
while son has not take out the garbage
    tell son to take out the garbage
    wait 2 minutes
    anger = anger + 1

A better example is from the previous homework. You may have wanted to be able to ask the user to enter a rate per hour and number of hours more than once. With a loop, you would be able to ask your questions over and over again, until the user wanted to stop.

while statement

In Python (as with many languages), a loop like this is implemented with the “while” statement

++++
Here is the generic form of the “while”

while <boolean expression>: # evaluates to True
        <indented block of code>

The syntax is very similar to the if statement. There is a Boolean expression, a colon, and an indented block of code. The difference is that using while statement, when the block of code is finished, control is transferred back to the while statement.

++++
Silly Python example:

```
looping = True
while looping ==  True:
        answer = input(
            "Please type the letter 'a': ")
        if answer == 'a':
                looping = False   # we're done
        else:
                print("Come on, type an 'a'! ")

print("Thanks for typing an 'a'")
```

Notice that we have an 'if' statement inside a while loop – perfectly fine. You can have any legal Python statements inside a while loop.

Also notice the way that I made this loop work. I started by setting a Boolean (looping) to True before the loop starts. This loop will continue as long as the Boolean is True, that is, as long as we continue to loop. Therefore, something inside the loop must affect the expression in the while statement. The while loop will continue until that condition changes.

Can write the while statement simpler:

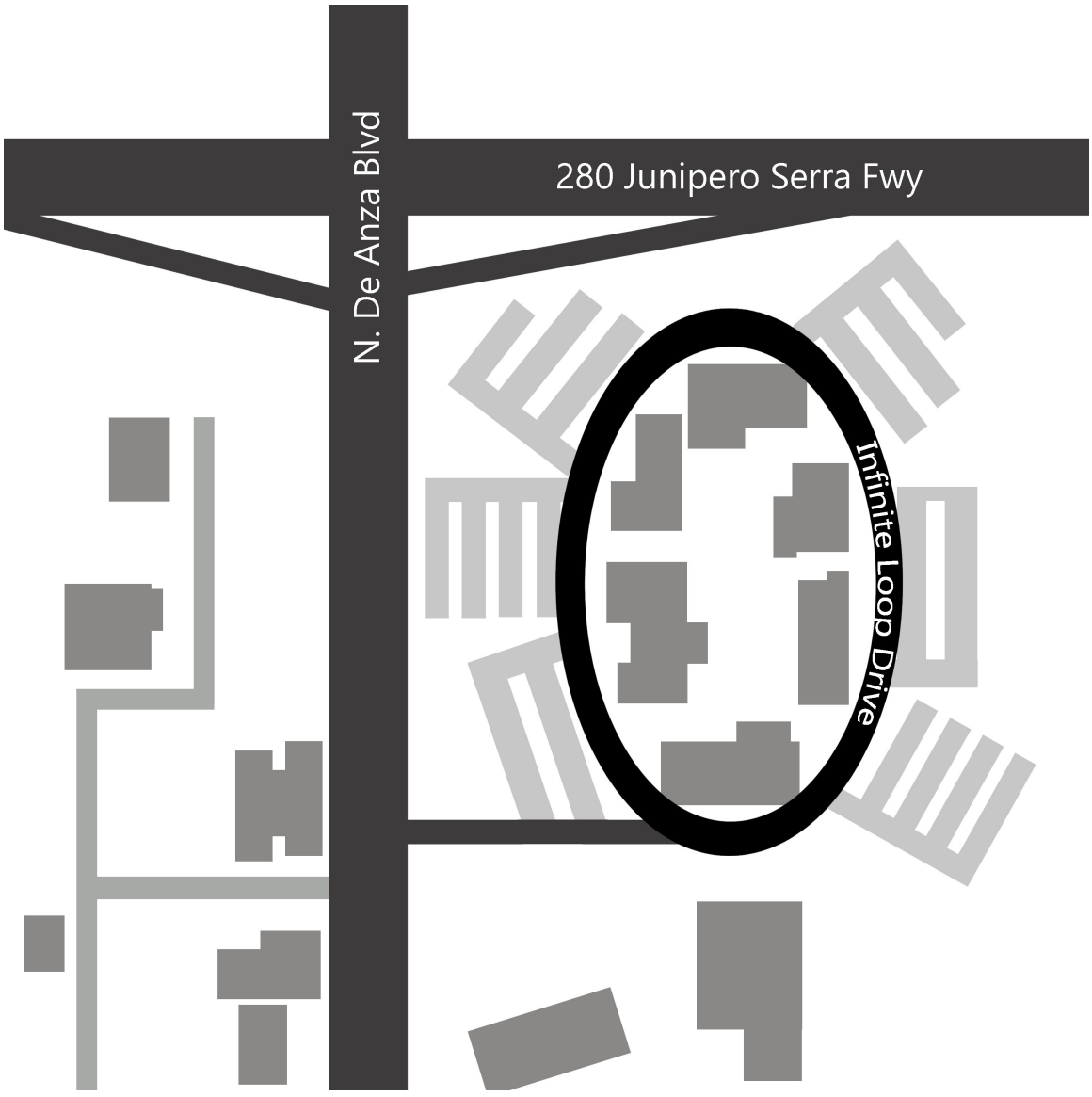
```
    while looping:
```

This will do the exact same thing. If you feel comfortable with this, you can write it this way. (I prefer this way.)

The Boolean expression in the while statement is often called the “exit condition” since that is what causes the loop to exit – the condition under which you exit the loop.

If nothing changes the value of the Boolean expression, then we have created what is called an infinite loop! That is, the loop will run forever.

++++ Apple’s corporate address is 1 Infinite Loop Drive:



First loop in a real program

++++Example: Let's build a simple program using a loop. The program ask the user for a target number, and goal of the program is calculate the sum of the numbers from 1 through the target number:

```
#Add up numbers from 1 to a target number

target = input('Enter a target number: ')
target = int(target)
total = 0
nextNumberToAddIn = 1
while nextNumberToAddIn <= target:
    # add in the next value
    total = total + nextNumberToAddIn
    print('Added in:' nextNumberToAddIn, 'total so far:', total)
    nextNumberToAddIn = nextNumberToAddIn + 1

print('The sum of the numbers 1 to', target, 'is:', total)
```

Notice the set up before the loop starts. We set total to 0, this will eventually be the total of all the numbers. And we set the variable nextNumberToAddIn to 1. This will be used to walk through the numbers from 1 to the number that the user entered.

Next, we build our while loop, and we loop as long as the nextNumberToAddIn is less than or equal to the target number.

Inside the loop, we add the nextNumberToAddIn to the total.

Also, we add one to the currentNumber to get to the next number. This is the key to exiting the loop. Remember we continue in the loop until the currentNumber is less than or equal to the target.

Increment and Decrement

In the above code we have this statement:

```
nextNumberToAddIn = nextNumberToAddIn + 1
```

++++DEF: Increment – when a variable adds to itself.

In the above statement, we say we are incrementing the variable. By default, we mean add 1 to the variable, but you can increment by any amount.

In many cases, you use this type of statement to count the number of times through a loop, or the number of tries you are doing something until there is a success or failure.

++++ A statement like this:

```
counter = counter + 1
```

can also be written as:

```
counter += 1
```

Gives the exact same result. C programmers are very used to this syntax. I will tend to use the longer syntax because it is more clear to me.

++++Note: The opposite of increment is “decrement”:

DEF: Decrement - when a variable subtracts from itself.

Again by default, when we use the word decrement, we imply that we subtract one from itself. But just like incrementing, you can decrement by any amount.

```
counter = counter - 1
```

Alternative syntax:

```
counter -= 1
```

In building our guessing game, we’ll use a counter and increment it with each guess to keep track of the number of number of guesses.

Running a Program a Number of Times

Let’s say we want to run the program multiple times. For example we want to allow the user to enter target numbers a number of times. To do that, we would build another loop.

To build this, we put the core of the program inside a function. Then we build a loop where we call the function, then ask if the user wants to go again. Here’s the code:

++++++ (Available in external file)

```
# Calculate total - repeated
```

```
def calculateSum(target):
    total = 0
    nextNumberToAddIn = 1
    while nextNumberToAddIn <= target:
        # add in the next value
        total = total + nextNumberToAddIn
        #increment
        nextNumberToAddIn = nextNumberToAddIn + 1
    return total
```

Let's say you want to run the whole program a certain number of times (let's say three times), you could do this as your main code:

```
nTimes = 0
while nTimes < 3:
usersTarget = input('Enter a target number: ')
    usersTarget = int(usersTarget)
    thisTotal = calculateSum(usersTarget)
    print('The sum of the numbers 1 to', usersTarget, 'is:',
thisTotal)
    nTimes = nTimes + 1    # increment the counter

print('OK Bye')
```

But if want, we can build a loop as the main code that runs as many times as the user wants to. The concept is that we would call our function do a run of the program, then ask the user if they want to try again. As long as the answer to the question is yes, then we loop back and start over.

```
goAgain = 'y'
while goAgain == 'y':
    usersTarget = input('Enter a target number: ')
    usersTarget = int(usersTarget)
    thisTotal = calculateSum(usersTarget)
    print('The sum of the numbers 1 to', usersTarget, 'is:',
thisTotal)

    goAgain = input('Do you want to try again (y or n): ')

print('OK Bye')
```

Python built-in packages

Back to our guess the number program. Next, we need to generate a random number. The real question is, how can a computer, which does everything exactly the same way, generate a random number. To answer this question, we have to learn more about how the Python world is put together.

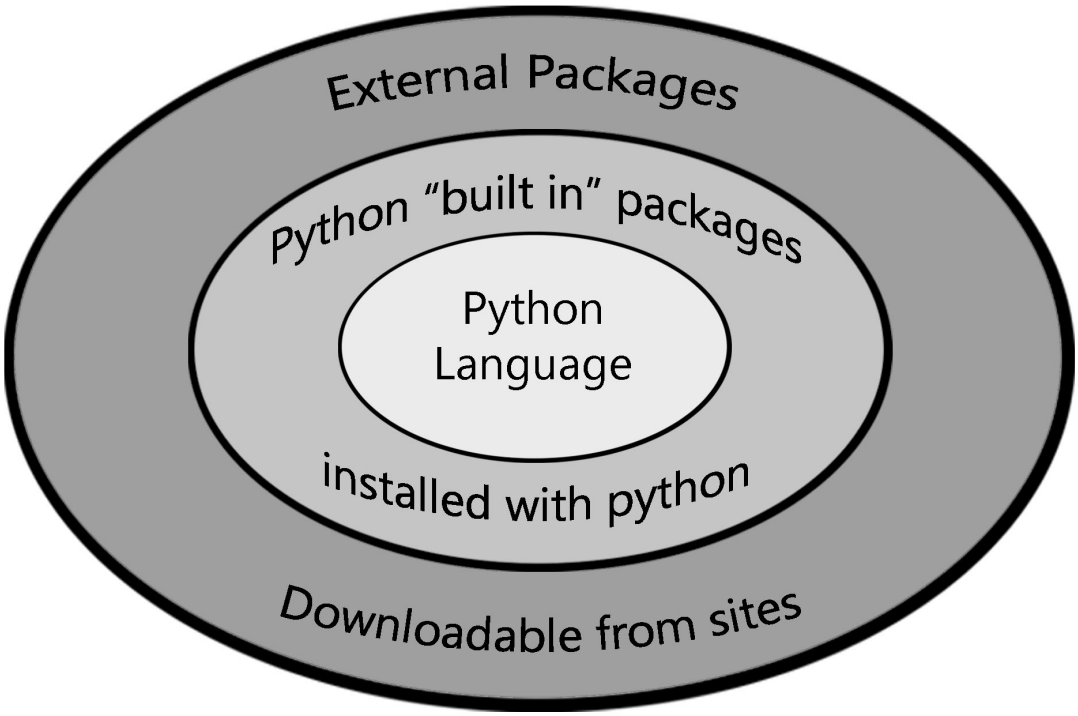
+++++
In order to keep programs small, the base Python language only has a small number of keywords (if, while, def, etc.) and built-in functions (int, str, input, etc.)

However, Python has some “built-in” packages of code that are available to programmers. (Sometimes called libraries.) The packages are installed on the hard disk of the computer when

you downloaded Python. This collection is commonly called the Python Standard Library.

One such package or library is called the random package. Some very smart programmer who worked on writing Python, wrote this package. It contains a number of functions that we can use to generate and use random numbers. But we don't get this package by default, because Python wants to keep your programs small.

++++ [Two more slides]



There are also external packages written by programmers all over the world, who make their code available to all

programmers. To get these packages, you need to download them from the internet. For example, if you remember in the first class, I showed you a game of Connect 4 that I wrote. That used a package called PyGame. There are thousands of these prebuilt packages available.

Generating a Random Number

But for now, we just want to use one of the built in packages, the random package, that ships with Python.

When you want to use a package, you need to tell Python that want to use it. You have to explicitly ask Python to include it in your program. The way you do this is to use the 'import' statement, which looks like this:

++++

import <packageName>

In this case, we want to import the random number package, so at the top, we need to add:

import random

Let's bring up the Python shell and enter that line.

This statement brings in a bunch of code into our program and makes all the functions in that package available to our program. You can see all the functions that are available by typing this:

help(random) [show]

This will give you a whole bunch of information about what the random package gives you.

There are many function calls that you can make, but for now, we are only interested in one: randrange

randrange accepts a number of parameters, but we will use it by passing in a low end value and a high end value. It returns a number in that range – including the low, but not including the high end. (We'll see this again later.)

++++
The way you call it is like this:

result = random.randrange(<lowValue>,
<upToButNotIncludingHighValue>)

You use the "random." to show that it is part of the random package. Then a period (called "dot"). Then randrange to say that you want to use that function

```
#random between 1 and 10
myRandomNumber = random.randrange(1, 11)

#random between 1 and 52
myRandomNumber = random.randrange(1, 53)
```

[Show example in shell]

Simulation of flipping a coin

Here is a very simple example program. We will simulate flipping a coin a large number of times. The program will run in a loop. Each time through the loop, we will randomly generate a 0 or 1. Then we'll do a mapping. We'll say that if we randomly got a 0, that means tails. If we got a 1, that means heads. We'll count the number of heads and tails. When the loop finishes, we'll report the results.

```
# Coin flip program
import random

nFlips = 0
nTails = 0
nHeads = 0

maxFlips = input('How many flips do you want to do? ')
maxFlips = int(maxFlips)

while nFlips < maxFlips:
    # Randomly choose 0 or 1, because a coin flip can only be in
one of two states (heads or tails)
    zeroOrOne = random.randrange(0, 2)

    # If we get a zero, say that was a Heads
    # If we get a one, we say that was a Tails
    if zeroOrOne == 0:
        nTails = nTails + 1
    else:
        nHeads = nHeads + 1

    nFlips = nFlips + 1
```

```
print()
print('Out of', nFlips, 'coin tosses, we had:', nHeads, 'heads,
and', nTails, 'tails.')
```

Notice that you don't directly randomly pick heads or tails. You randomly pick from a range that encompasses all possible outcomes, then you map the numeric answer to the outcomes you were looking for. In this case, there are only two possible outcomes, so we ask for random numbers between 0 and 1, and map tails to zero, and heads to 1.

Another example of this would be if you were writing a game to play rock, paper, scissors. In this case, you have three possible choices, so you would generate a random number between 0 to 2 (or even more humanlike, 1 to 3). Then you would map the random number you got back to rock, paper, or scissors (typically with an if/elif/else statement).

[If there is time, show Dragon program]

Sample Dragon program

Show dragon.py example. Uses:

- print statement
- variables (global and local)
- functions
- arguments and parameters
- return statement
- while loop

if/else statement
application of truth tables inside an “if” expression
importing a Python module, e.g., random

Modify the program to have four caves (that is, allow the user to enter 1, 2, 3, or 4). Randomly choose one of the four caves as the “friendly” cave” (“Gives you his treasure”). If the user did not choose the friendly cave, have the program output one of three random messages, such as:

Buys you a pizza
Slaps you with its tail
Takes you to a movie

Creating an infinite loop

When I introduced the while statement, I said that as long as the Boolean expression evaluates to True, then the while statement would continue to loop. The loop only stops when the Boolean expression evaluates to False.

Earlier, we said that that he Boolean expression in the while statement is referred to as the exit condition. That is, the test for exiting the loop is done in the while statement. Here is an example:

```
looping = True
while looping:
    <statement(s)>
    if <found exit condition?>:
        looping = False     # found the exit condition at this point
```

```
    else:
        <continuing statement(s) inside the loop>
```

In effect, you have found the exit condition, but you dont't exit the loop until execution goes back to the while statement. Unfortunately, this style often makes it more difficult to write the continuing part of the loop that follows. The code that you run if the exit condition has not been reached must get indented.

If you need to detect and handle multiple exit conditions, each if statement would set the same variable (that is later checked in the while statement) and the code that continues the normal execution gets indented further. This excessive indenting makes it difficult to write and even more difficult to read through the normal path through the loop.

Fortunately, there is another way to build a while loop.

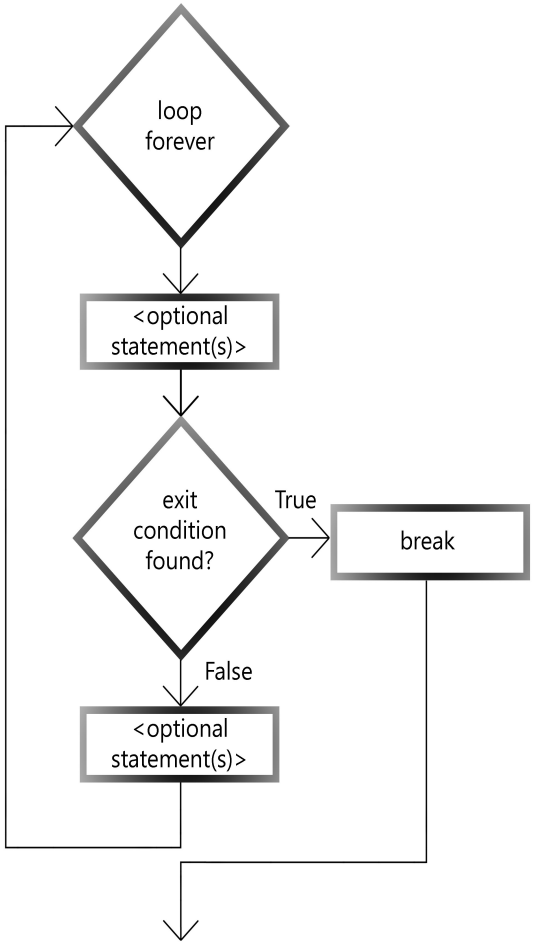
A new style of building a loop, and the break statement

Earlier we talked about how you might accidentally create an infinite loop. The simplest form of an infinite loop would be like this:

```
while True:
    <statement(s)>
```

This would run forever. However, Python provides another statement called the break statement, made up of just the word "break". If you are in the middle of a loop, and you find

some condition where you want to exit the loop immediately, you can use a “break” statement. Here is an flowchart:
+++++



When the break statement executes, control will transfer to the first statement past the end of the loop – it exits or breaks out of the loop. We’ll use this in our full game.

++++ Example:

```
while True:
        line = input(‘Type done to exit’)
        if line == ‘done’:
                break
        print(‘You entered:’, line)
```

```
print('Finished')
```

If you are familiar with the C language, this is effectively a go to statement, but instead of going to a label, it goes directly to the first statement past the end of the loop.

In our Guess the Number game, we will have two exit conditions. First if the user guesses the correct answer, and second if the user runs out of guesses. We'll use a break statement in each of those cases.

Asking if the user wants to repeat, the empty string

Final thing before we get to the guessing game. If you want to build a loop where you ask the user if they want to continue, you can write something like this:

```
while True:
    # do whatever
    # now ask to go again
    goAgain = input
      ('Press Return/Enter to quit, or anything else to continue')
    if goAgain == '':
      break
```

If the user just presses the Return key (Mac) or the Enter key (PC), then input generates the empty string. That is, a string with no characters. This is represented as '', which we read as

quote quote. In this case, the variable goAgain is set to the empty string, and the loop continues.

This is a very useful technique. Almost all the loops that I write work this way. You create a infinite loop using a “while True:” statement, but inside the loop you look for the ending condition – when you find it, your break out of the loop.

Now we can look back at the “Magic 8-Ball” program that I showed on the first day. You should now be able to understand everything in that program.

Rather than jumping right into the code of our number guessing game, let’s talk about how the program is going to work. We’ll come up with an overall approach, before writing the actual code.

To do this, let me add another more definition

Pseudo-code

++++

DEF: Pseudo-code – an English-like description of an algorithm in a made up computer language

Very often, programmers will develop an algorithm, and write the algorithm in pseudo-code before writing it in a real computer language. This allows programmers to think through a problem without having to worry about the detailed syntax of a computer language.

To show you what I mean, here is the algorithm for our 'guess the number' program, written in pseudo-code.

++++

Show introduction
Choose random target
Initialize a guess counter

Loop forever
    Ask the user for a guess
    Increment guess counter
    If user's guess is correct, we're done
    If user's guess is too low, tell user
    If user's guess is too high, tell user
    If reached max guesses, tell answer, we're done.

We can take this pseudo-code, and turn each statement into a comment inside IDLE.

Copy from here, paste into new document in IDLE. In IDLE, select all, do a File -> Comment out region. (Not sure why IDLE adds two number signs '##' for each comment line.)

It can sometimes be difficult to build and test a program that has randomization in it, because it will typically do different things on each run. So, to start building our program, let's start by hard-coding a target number of 10. Once we get the logic working with a known target number, we'll modify the

code to pick the target number randomly. We'll build the basic logic and test it to ensure that it is working correctly.

Now we can write the full code of the program in Python.

++++

```
# Guess the Numbers

import random     # include everything in the "random" module

#Show introduction
print('Welcome to my guess the number program')
print('Guess my number between 1 and 20')
# Choose random target
target = random.randrange(1, 21)  # start with:  target = 10
# Initialize a guess counter
counter = 0  #  keeps track of the number of guesses


while True:

   #Ask the user for a guess
   guess = input('Take a guess: ')  #prompt the user to enter a number
    guess = int(guess)   # convert the characters to an integer
    #  Increment guess counter
    counter = counter + 1

    # If user's guess is correct, we're done
    if guess == target:
        print('You got it in ', counter, ' guesses')
```

```
        break   # exit the loop now

    #  If user's guess is too low, tell user
    elif guess < target:
        print('Your guess is too low')

    #  If user's guess is too high, tell user
    else:
        print('Your guess is too high')

    #  If reached max guesses, tell answer, we're done.
    if counter == 5:
        print('You didn't get it in 5 guesses - loser!')
        print('The number I was thinking of was: ', target)
        break   # we're done

print('Thanks for playing')
```

Playing the game multiple times

Finally, we may want to allow the user to play our game multiple times. To do that, we take the core of the program we wrote, and put that inside a function. Let's call that function 'playOneRound'. [Show how to indent a number of lines at once.]

Then the main code becomes a small loop. In the loop, we call our function to play the game, then ask the user if they want to play again. Here's how we do that part:

```
while True:
    playOneRound()
    print()
```

```
    goAgain = input('Do you want to play again, press ENTER to
continue, or q to quit.')
    if goAgain == 'q':
        break
```

Modify the program to use constants.

In the program we said that the user will have 5 tries to guess the number. We also said that the target number will be between 1 and 20. Notice that we have those numbers explicitly written in our code. This works, but if we ever wanted to change our program, we would have to read through every line looking for those numbers, ensure that they should be changed, and change every occurrence. Instead of doing that, let's modify the code to add two constants:

```
MAX_GUESSES = 5
MAX_VALUE = 20
```

Then change the code to match. This allows us to change the rules of the game by only changing a value in a single place.

Error detection and correction

Our program is working perfectly ... as long as the user enters an integer. But what happens if the user makes a mistake – either on purpose or not. When asked to type in an integer, if the user types in something that is not an integer, (for example, the string 'abc'), the program crashes with this error

ValueError: invalid literal for int() with base 10: 'abc'

That's a little hard to read, but what it is really saying is that the line:

guess = int(guess)

could not convert what the user typed in to an integer.

It's rather a rude way for the program to behave. Let's see how we can detect this type of error, and figure out a way to recover from it. It's going to involve two new concepts.

Error checking with try/except

Python provides a mechanism for doing a check for an error, but it takes a little getting used to. You can use it wherever you want, but it works especially well within a while loop. Here's the idea.

We ask the user to enter a number. But before we try to convert the number to an integer, we ask Python to watch what's going on, and let us know if there is a problem. If there is no problem, we go on with what we wanted to do, if there is a problem with the conversion, then we can print an error message for the user.

This is implemented in Python with a "try/except" block. Here's what it looks like:

```
try:
   <some statement(s) that may cause an error>
except:
```

```
    <statement(s) to execute if an error occurs>
```

Here is an example of how it might be used:

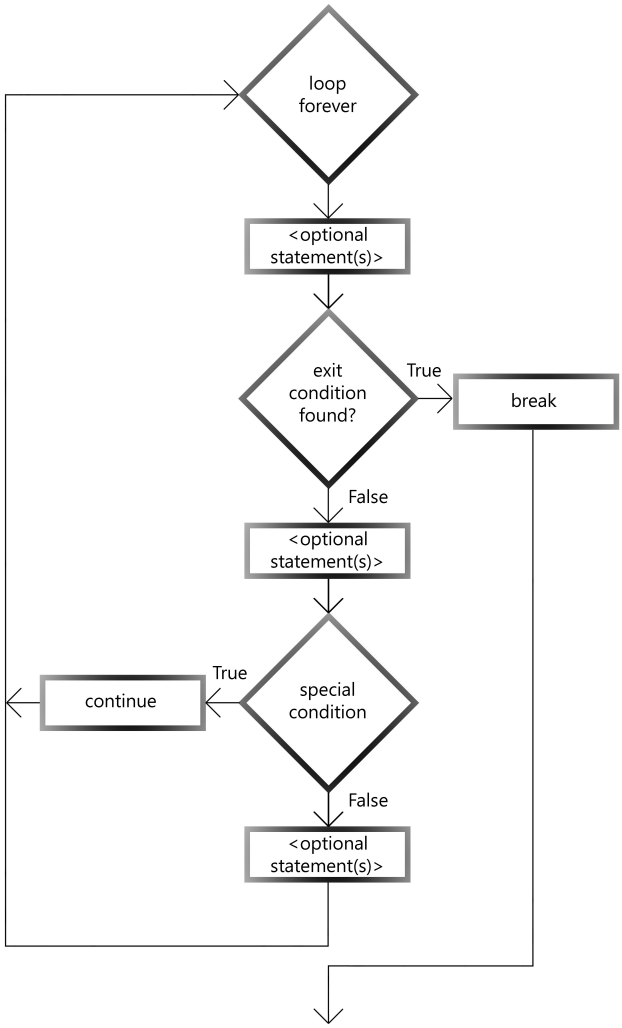
```
userNumber = input('Please enter an integer: ')
try:    # attempt to convert to an integer:
    userNumber = int(userNumber)
except:   # runs only if an error happened
    print('Sorry, that was not an integer')
```

Let's add that in our program and see how it works.

The good news is that it "catches" the error and prints an appropriate error message. The bad news is that after printing an error message, the code tries to use the value of our variable, assuming that it is an integer, and fails a little later, (specifically when trying to compare the integer to the string that the user typed). So, we need one more piece.

continue statement

What we really want, is a way to go back to the top of the loop and ask the loop question again, without executing the rest of the code inside of the loop. To transfer control, we can execute a new statement called the "continue" statement. Let's take look at a flowchart:



Here's how we can use try/except in our Guess the Number program to ensure that the user enters a valid integer:

```
def playOneRound():
    #Choose random target
    target = random.randrange(1, MAX_VALUE + 1)

    #Initialize a guess counter
    guessCounter = 0
```

```
#Loop forever
while True:
    #Ask the user to for a guess
    userGuess = input('Take a guess: ')

    # Check for potential error
    try:
        userGuess = int(userGuess)
    except:
        print('Hey, that was NOT an integer!')
        continue

    #Increment guess counter
    ...
```

Again, if the code detects an error trying to convert the users guess, the then except block will run. After giving an appropriate error message, the continue statement passes control back to the while statement, and the loop starts over. Now we have a way of detecting and correcting user errors without crashing.

[If there is time ...]

Building error checking utility functions:

Here's how we can use this concept whenever we ask the user to enter a number. The idea is that we will create a function whose purpose is to return an integer. If the user types something that is not an integer, then we will give an error, and ask the user to enter an integer again. When the user types an

integer, the function returns the integer value that the user entered.

Here's some code to handle it.

```
def getIntegerFromUser(prompt):
    while True:
        number = input(prompt)
        try:
            number = int(number)
        except:
            print('That is not an integer, please try again.')
            continue
        #  everything OK
        return number

def getFloatFromUser(prompt):
    while True:
        number = input(prompt)
        try:
            number = float(number)
        except:
            print('That is not a float, please try again.')
            continue
        # everything OK
        return number


myInteger = getIntegerFromUser('Please enter an integer: ')
print(myInteger)


myFloat = getFloatFromUser('Please enter a float: ')
print(myFloat)
```