Flow Control

Kelly M. Thayer

Week 5 Lecture 4

2017-02-21

Contact info:
kthayer@wesleyan.edu
Office Hours: M 8:30-10:30 AM, T 10:10-11:15 AM, W 5:00-6:00 PM
Office: Exley 322/325

Announcements

- Upcoming Exam: Tuesday March 7, 2016
- If you have an academic accommodation, please present your letter to your instructor this week (if you have not done so already)

Learning Objectives

- A few useful things
  - User input
  - Augmented assignments
  - Accumulators
- Understand how to affect the flow of a program **if** some condition is met and write the syntax
  - Execution with if statements (5.4)
  - Execution with if… else statements (5.5)
  - Chained conditionals (5.6)
  - Nested conditionals (5.7)
- Understand how to affect the flow of a program **while** some condition is met and write the syntax
  - Basic while statements (7.3)
  - Stopping execution with break (7.4)
  - Square Roots example (7.5)

A few useful things before we get started…

How to get user input

- Use the **`input`** function to obtain user input
- Input takes input
  - In brackets and quotes, you can specify a prompt to display to the user
- The program pauses to wait for the user to type the input on the terminal
- Set a variable equal to the output of the **`input`** function to capture the user's answer

Syntax:

<variable> = **`input`** (' prompt text')

Example:

```
today = input ('What day of the week is it?')
print('You said today is ',today)
```

```
>>>
==== RESTART: F:\09_Python_course  S17\01_Lecture\lecture_04_If_while.py ====
What day of the week is it?Monday
You said today is  Monday
>>>
```

How to get user input: Getting Numerical Input

The problem: input is considered to be a string

```
x = input('please enter an integer.')
print('The type of the input,', x, 'is ',type(x))
```

```
==== RESTART: F:\09_Python_course  S17\01_Lecture\lecture_04_If_while.py ====
please enter an integer.3
The type of the input, 3 is  <class 'str'>
```

How to make the type be integer:

```
x = int(input('please enter an integer.'))
print('The type of the input,', x, 'is ',type(x))
```

```
==== RESTART: F:\09_Python_course  S17\01_Lecture\lecture_04_If_while.py ====
please enter an integer.4
The type of the input, 4 is  <class 'int'>
```

Augmented Assignment

- Shorthand for common patterns of incrementing, decrementing, multiplying , and dividing a variable
- Some examples:
  - x = x + 1 -> x+=1
  - x = x – 1 -> x-=1
  - x = x * 2 -> x*=2
  - x = x / 2 -> x/=2

Accumulators

- Accumulate information over the course of multiple iterations, such as in a loop
- An example is a running total
- We will see other examples in other loop contexts later in the course
- Can take the form of augmented assignment
- Must initialize them before use

```
def run_sum():
    # Signature: (no parameters) -> float
    # precondition: x is a number or 'done'
    # computes running sum
    acc=0
    while True:
        x = input('Enter number to sum, done to exit.')
        if x == 'done':
            print('exiting loop')
            break
        else:
            acc += float(x)

    return acc
```

Accumulator initialization

Augmented assignment

If Statements

Conditional Execution

- When something is to be done *contingent upon* some condition
- Use Boolean type to decide
- Use an **`If`** statement
- A branch guard determines whether it will run
  - If the Boolean evaluates to True, the branch will run
  - Otherwise, it will not run

Conditional Execution: Syntax & Ex.

Syntax:

```
if <boolean_expression>:
        <first conditional statement>
        .
        .
        .
        <last conditional statement>
```

Branch guard – the Boolean expression

Branch – the indented block to be executed

Example:

```
if x < 0:
    print('x is negative')
    xsq = x * x # compute the square
    print('and the square is positive')
    print('x squared is ',xsq)
```

Conditional Execution Example

- It's 8:45AM. Should I go to Shanklin 107 for COMP112 lecture?
  - Well that depends if it's a Tuesday or not.
- Let's write a conditional (in English)
  - If today is Tuesday, then go to Shanklin 107 for COMP112 lecture
  - If {condition is True} then {result}

Sounds like Boolean expressions… could they be useful here?

Recall: Boolean Expressions

- The value of Boolean expressions is either **`True`** or **`False`**
  - 5==5 #True
  - 4==5 #False
- **`True`** and **`False`** have type **`bool`**
- Use relational operators
  - == # equal
  - != # not equal
  - > # greater than
  - < # less than
  - >= # greater than or equal to
  - <= # less than or equal to

The condition of an if statement is expressed as a Boolean expression.

Use a Boolean Expression to Write a Conditional

- English: If today is Tuesday, then go to Shanklin 107 for COMP112 Lecture
- Write the corresponding code:

```
>>> today = 'Tuesday'
>>> if today == "Tuesday":
        print("It's your lucky day!  Comp 112 lecture SH107, yay!")


It's your lucky day!  Comp 112 lecture SH107, yay!
```

When the Condition Is Met
vs.
When It Is Not Met

```
>>> today = 'Tuesday'
>>> if today == "Tuesday":
        print("It's your lucky day!  Comp 112 lecture SH107, yay!")


It's your lucky day!  Comp 112 lecture SH107, yay!
>>> today = 'Wednesday'
>>> if today == "Tuesday":
        print("It's your lucky day!  Comp 112 lecture SH107, yay!")


>>>
```

The Body of a Conditional

- May have one or more statements; no max number
- Be sure to indent all statements belonging to the conditional
- If you don't have a statement, use **`pass`**.

Example:

```
def day():
    today = input('What day is today? Monday, Tuesday,..., Sunday')

    if today == 'Tuesday':
        print("It's your lucky day!")
        print('COMP112 Lecture at 107 Shanklin, yay!')

    if today != 'Tuesday':
        pass # don't do anything
```

```
==== RESTART: F:\09_Python_course  S17\01_Lecture\lecture_04_If_while.py ====
>>> day()
What day is today? Monday, Tuesday,..., SundayTuesday
It's your lucky day!
COMP112 Lecture at 107 Shanklin, yay!
>>> day()
What day is today? Monday, Tuesday,..., SundaySunday
>>>
```

Alternative Execution

- Suppose you want to have a different outcome based on whether a number is odd or even.
- Using what you've learned so far, chat with your neighbor to come up with a Boolean expression to determine if a number is odd or even. Then incorporate it into an **`if`** statement.

Recall: The Modulus

- % operator yields the remainder from division of the first argument by the second
- If a number is even, then it is evenly divisible by 2
- It follows that if the number is even, then the remainder when divided by 2 is 0
- This can be written as an if statement

```
def odd_or_even(n):
    # signature: int -> nonetype
    # assesses if n is odd or even
    remainder = n%2
    if remainder == 0:
        print(n, 'is even')
    else:
        print(n, 'is odd')
```

```
==== RESTART: F:/09_Python_course  S17/01_Lecture/lecture_04_If_while.py
>>> odd_or_even(2)
2 is even
>>> odd_or_even(7)
7 is odd
>>>
```

Alternative Execution: `if-else`

- When something is to be done *contingent upon* some condition, and if the contingency is not met, do something else
- Again, use Boolean type to decide
- Add an **`else-clause`** to form an **`if-else`** statement
- **`else`** must be associated with an if-block
- A branch guard determines whether it will run
  - If the Boolean evaluates to True, the branch associated with it will run
  - Otherwise, the branch with the else-clause will run
- Exactly one branch is run

Alternative Execution: Syntax

Syntax:

Branch guard – the Boolean expression

```
if <boolean_expression>:
        <first conditional statement>
        .
        .
        .
        <last conditional statement>
else:
        <first conditional statement>
        .
        .
        .
        <last conditional statement>
```

True branch

False branch

Alternative Execution: Example

Example:

```
# slide 21
def alt_exec():
    # signature: no input param, -> nonetype
    x = int(input('Please enter an integer.'))

    if x < 0:
        print('x is negative')
        xsq = x * x # compute the square
        print('and the square is positive')
        print('x squared is ',xsq)
    else:
        print('x is non-negative')
```

```
==== RESTART: F:\09_Python_course  S17\01_Lecture\lecture_04_If_while.py ====
Please enter an integer.-2
x is negative
and the square is positive
x squared is  4
>>>
```

Chained Conditionals: if-elif…elif

- When something is to be done *contingent upon* some condition, and if the contingency is not met, check if other contingencies are met
- Again, use Boolean type to decide
- Add multiple **`elif-clauses`** to form multiple branches.
- A branch guard determines whether it will run
  - The first Boolean that evaluates to True will have the branch associated with it will run
  - Otherwise, the next condition is evaluated. Evaluation occurs in order.
- At most, one branch is run
  - If multiple guards are true, the first which is true is the one that is executed
  - If no guards are true, then none of the branches are executed.

Chained Conditionals: Syntax

```
if <boolean_expression_1>:
       <first conditional statement>
       .
       .
       .
       <last conditional statement>
elif <boolean_expression_2>:
       <first conditional statement>
       .
       .
       .
       <last conditional statement>
…
elif <boolean_expression_n>:
       <first conditional statement>
       .
       .
       .
       <last conditional statement>
```

Branch guard – the Boolean expression

Branch 1
Executes if Boolean expression 1 is true

Branch 2
Executes if Boolean expression 1 is false and Boolean expression 2 is true

Branch n
Executes if Boolean expressions above are false but Boolean expression n is true

Chained Conditionals: Example

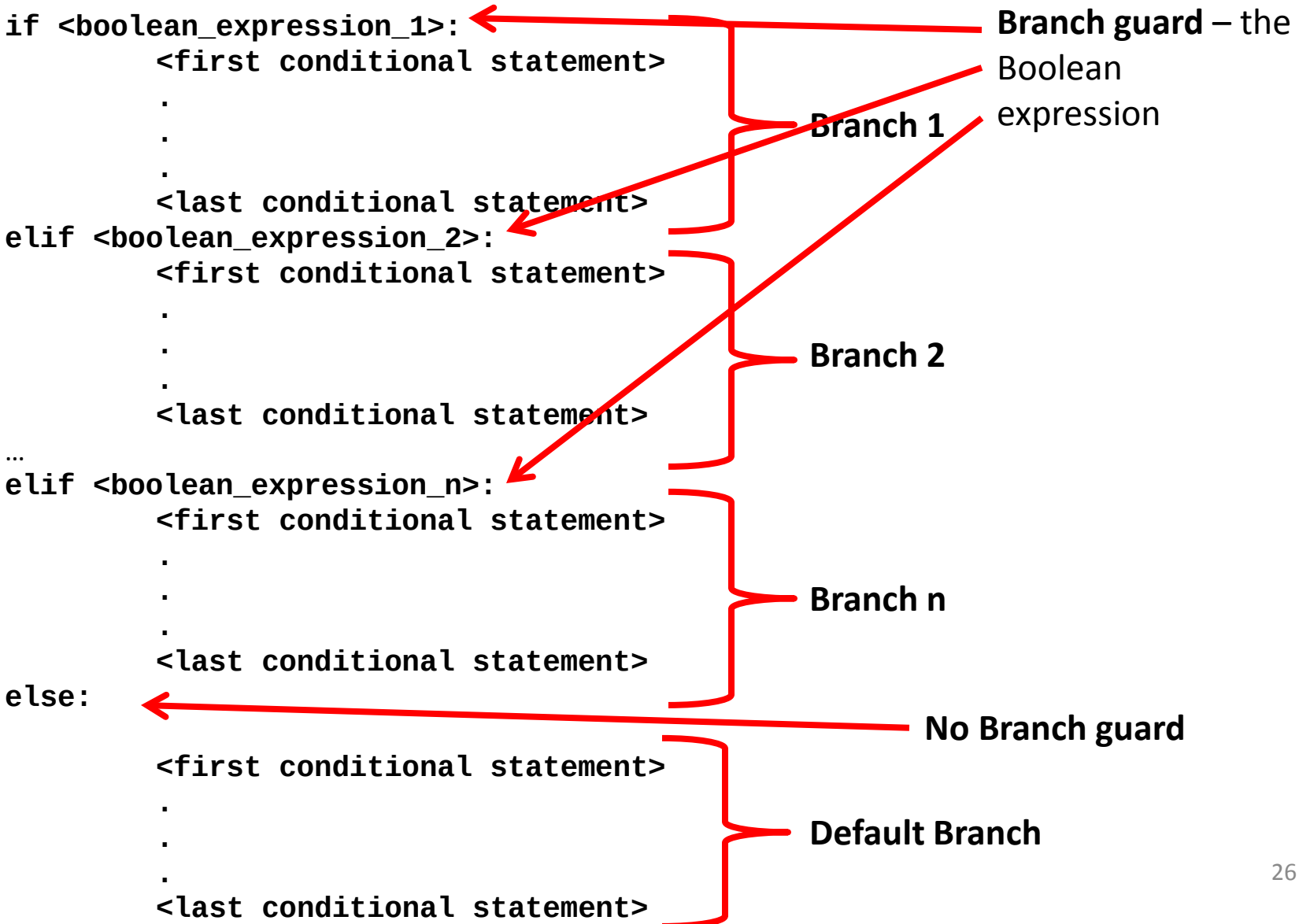
```
# Not chained
if x > 3:
    print('x is greater than three')
if x > 2:
    print('x is greater than two')
if x > 1:
    print('x is greater than one')

# chained
if x > 3:
    print('x is greater than three')
elif x > 2:
    print('x is greater than two')
elif x > 1:
    print('x is greater than one')
```

What will the outputs be for x = 5? How will they differ?

Multiple Alternatives: If-Elif…Elif…Else

- When something is to be done *contingent upon* some condition, and if the contingency is not met, check if other contingencies are met
- Again, use Boolean type to decide
- Add multiple **`elif-clauses`** to form multiple branches.
- Add an else-clause to catch whatever does not meet the prior conditions
- Execution operates similar to the examples shown prior

Multiple Alternatives: Syntax



Multiple Alternatives: Example

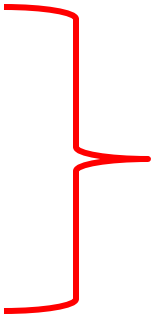
```
if x > 3:
    print('x is greater than three')
elif x > 2:
    print('x is greater than two')
elif x > 1:
    print('x is greater than one')
else:
    print('None of the above apply')
```

Nested Conditionals

- Sometimes more than one condition needs to be met in order to arrive at the final answer
- More calculation may be required after one decision but before another decision
- Use conditionals as part of the body of another conditional
- A nested **`if`**-statement becomes one of the conditional statements of another **`if`**-statement.
- The beginning of each If-block must begin with **`if`** (and not **`elif`**)
- No limit on depth of nesting

Nested Conditionals: Syntax

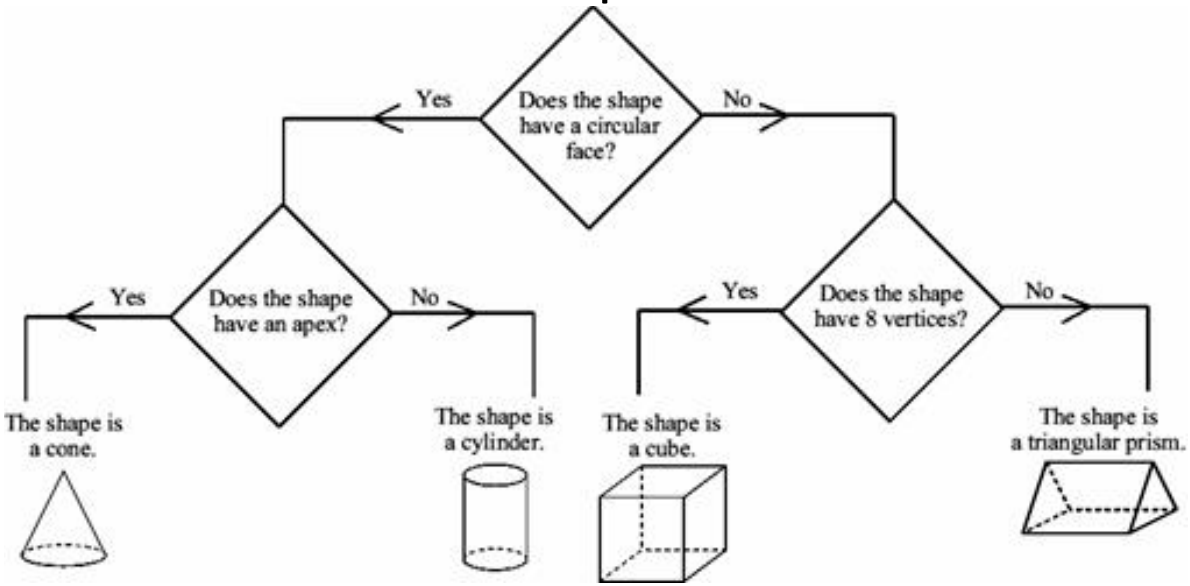
```
if <boolean_expression_1>:
        <conditional statement 1.1>
        …
        if <boolean_expression 1A>:
                <conditional statement 1A.1>
                .
                .
                .
                <conditional statement 1A.n>
        …
        <conditional statement 1.n>
elif <boolean_expression_2>:
        <first conditional statement>
        .
        .
        .
        <last conditional statement>
…
else:

        <first conditional statement>
        .
        .
        .
        <last conditional statement>
```



Nested Conditional

Nested Conditionals: Example

Classification of Four Geometric Shapes



- Write nested if-statements to classify the four geometric solids as suggested by the diagram. Prompt the user with yes or no questions to obtain the answers.
- How could the diagram be extended to include such solids as the rectangular prism, hexagonal prism, sphere, etc.?

Nesting: Example

```
def classify_shape():
    YorN = input ('Does the shape have a circular face? (Y or N)')
    if YorN == 'Y':
        YorN2 = input ('Does the shape have an apex? (Y or N)')
        if YorN2 == 'Y':
            print('The shape is a cone.')
        elif YorN2 == 'N':
            print ('The shape is a cylinder.')
        else:
            print ('That is not a valid Yes or No answer.')
    elif YorN == 'N':
        YorN2 = input ('Does the shape have 8 vertices? (Y or N)')
        if YorN2 == 'Y':
            print ('The shape is a cube.')
        elif YorN2 == 'N':
            print ('The shape is a triangular prism.')
        else:
            print('That is not a valid Yes or No answer.')
    else:
        print('That is not a valid Yes or No answer.')
```

Loops

Repeated Execution: While Loops

- If statements execute a code block one or zero times
  - This depends if the guard is satisfied
  - If multiple guards are true, the first will execute (and the other(s) will not)
- While statements execute a code block any number of times, as long as the guard is satisfied
  - No limit as to how many times it will execute
  - If the guard is not satisfied, the associated block will not execute
    - 0 times is possible
  - It may execute an infinite amount of times, called an infinite loop, if the guard never stops being satisfied
    - This is undesirable and should not be done on purpose
    - Occasionally you might make a mistake and this could happen
    - Use control-C to exit

Repeated Execution: While Loops

Syntax:

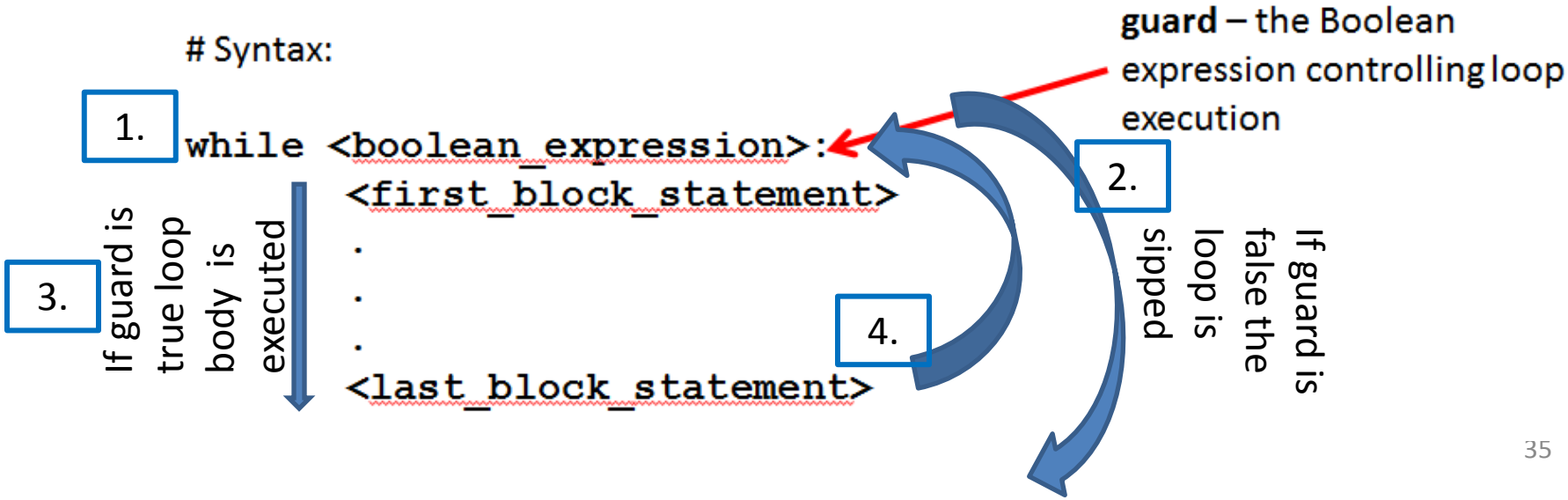
```
while <boolean_expression>:
        <first_block_statement>
        .
        .
        .
        <last_block_statement>
```

guard – the Boolean expression controlling loop execution

Repeated Execution: While Loops

Steps in flow control for a **`while`** loop

1. The guard is evaluated
2. If the value of the guard is False, then the body is skipped
3. If the value of the guard is True, then the body is executed
4. After the last statement of the body, control returns to the guard, and the process is repeated



Repeated Execution: While Loop Example

```
def count_to_20_integer(n):
    # Signature: int -> nonetype
    # counts up to the first multiple over 20
    x = 0 # initialization
    count=0
    while x < 20:
        x += n # augmented assignment
        print(x)
        count=count+1
    print('The loop executed ',count,'times.')


==== RESTART: F:\09_Python_course
>>> count_to_20_integer(6)
6
12
18
24
The loop executed  4 times.
>>>
```

Iteration with While Loops: Factorial Calculation

```
def calc_factorial(n):
    # signature: int -> int
    # precondition n>=0
    # computes the factorial n!

    acc = 1
    while n > 0:
        acc *= n # acc = acc * n
        n -= 1   # n = n - 1
    print('The factorial is',acc)
    return acc
```

Recall: factorials

4! = 4 * 3 * 2 * 1 = 24

n! = n * (n-1) * (n-2) * … * 1

```
==== RESTART: F:\09_Python_course  S17\01_Lecture\lecture_04_If_while.py ===
>>> calc_factorial(4)
The factorial is 24
24
```

Repeated Execution Terminated with **`break`**

- Sometimes the condition to break the loop occurs at some point during the execution of the loop
- **`Break`** provides a means of ending the loop
- Often used inside an **`if`** statement

Syntax sample:

```
while <boolean_expression>:
        <first_block_statement>
        …
        if <condition>:
                <first conditional statement>
                …
                break # execution of loop ends
        …
        <last_block_statement>
```

Example with **Break**

```
def run_sum():
    # Signature: (no parameters) -> float
    # precondition: x is a number or 'done'
    # computes running sum
    acc=0
    while True:
        x = input('Enter number to sum, done to exit.')
        if x == 'done':
            print('exiting loop')
            break
        else:
            acc += float(x)

    return acc
```

```
==== RESTART: F:\09_Python_course  S17\01_Lecture\lecture_04_If_while.py ====
>>> run_sum()
Enter number to sum, done to exit.3
Enter number to sum, done to exit.4
Enter number to sum, done to exit.done
exiting loop
7.0
```

Writing a Function: Putting it All Together

Newton's Sqrt

- Write a function that computes the square root of a number
- Use Newton's square root formula
  - sqrt_est = (guess + a/guess)/2
    - sqrt_est is the estimate of the square root as computed by Newton's formula
    - Guess is some guess as to what the answer might be.
  - Iteratively refine the sqrt_est by taking the answer from the previous iteration and make it the guess for the current iteration
  - Keep improving the guess with more iterations until the sqrt_est is within 0.5 of the last guess. (tolerance)
  - Begin the iterative process with a user specified guess
- Should compute square root only for values greater than or equal to zero; check the user input to verify that this is true
- Report how many iterations were computed.
- Return the best estimate of the square root.

Putting it together: Newton's Sqrt

```
def newton_sqrt(a):
    # Signature: float -> float (or nonetype on error)
    # precondition: a >=0 and real
    # Newton's square root method

#...check the input
    if a < 0:
        print('Error: Only computes sqrt for a>=0.')
        return

#...initialization
    count = 0
    guess = float(input('Please guess the square root.'))
    sqrt_est = guess + 1

#...the while loop
    while abs(guess - sqrt_est)>= 0.5:
        guess=sqrt_est # make the current guess the calc from last time
        sqrt_est = (guess + a/guess)/2 # Newton's sqrt formula
        print('The estimated square root is ',sqrt_est)
        count += 1
#...finishing up
    print('The square root estimate has converged in ', count, 'guesses')
    return sqrt_est
```

Newton's Sqrt

```
==== RESTART: F:\09_Python_course  S17\01_Lecture\lecture_04_If_while.py ====
>>> newton_sqrt(4)
Please guess the square root.3
The estimated square root is  2.5
The estimated square root is  2.05
The square root estimate has converged in  2 guesses
2.05
>>> newton_sqrt(4)
Please guess the square root.20000
The estimated square root is  10000.500099995
The estimated square root is  5000.250249987499
The estimated square root is  2500.1255249737305
The estimated square root is  1250.0635624466993
The estimated square root is  625.0333811419938
The estimated square root is  312.5198904000946
The estimated square root is  156.26634479271783
The estimated square root is  78.14597105753354
The estimated square root is  39.09857865881474
The estimated square root is  19.60044208402417
The estimated square root is  9.902259556828552
The estimated square root is  5.153103882256142
The estimated square root is  2.9646675401336853
The estimated square root is  2.1569456693524582
The estimated square root is  2.0057099127433946
The square root estimate has converged in  15 guesses
2.0057099127433946
>>>
```

A few loose ends

Why do we use functions?

- Modularity
- Divide and conquer
- Readability of code

THE END