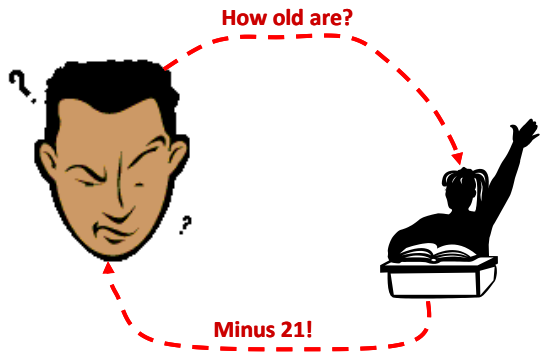
Loops In Python

In this section of notes you will learn how to rerun parts of your program without having to duplicate the instructions.

Repetition: Computer View

- Continuing a process as long as a certain condition has been met.

Ask for age as long as the answer is negative

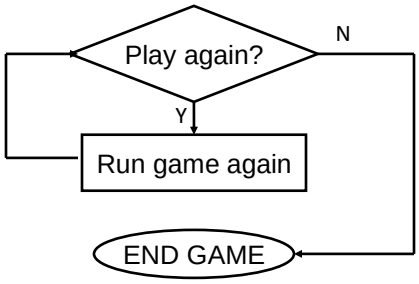


How To Determine If Loops Can Be Applied

- Something needs to occur multiple times (generally it will repeat itself as long as some condition has been met).
- Example 1:

Re-running the entire program

Flowchart



Pseudo code

While the player wants to play

    Run the game again

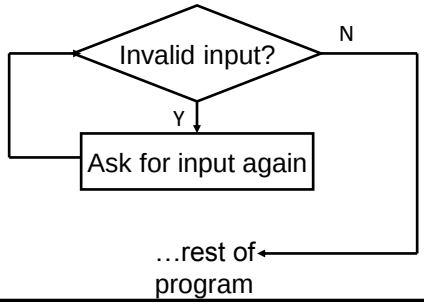
How To Determine If Loops Can Be Applied (2)

- Example 2:

```
Enter your age (must be non-negative):  -1
Enter your age (must be non-negative):  27
Enter your gender (m/f):
```

Re-running specific parts of the program

Flowchart



Pseudo code

While input is invalid

    Prompt user for input

Basic Structure Of Loops

Whether or not a part of a program repeats is determined by a loop control (typically the control is just a variable).

- Initialize the control to the starting value
- Testing the control against a stopping condition (Boolean expression)
- Executing the body of the loop (the part to be repeated)
- Update the value of the control

Types Of Loops

1.Pre-test loops

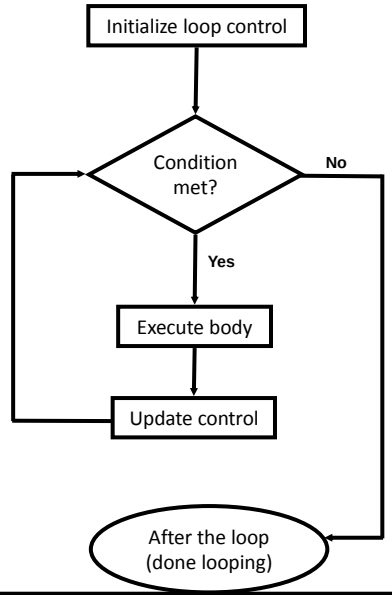
- Check the stopping condition *before* executing the body of the loop.
- The loop executes *zero or more* times.

2.Post-test loops

- Checking the stopping condition *after* executing the body of the loop.
- The loop executes *one or more* times.

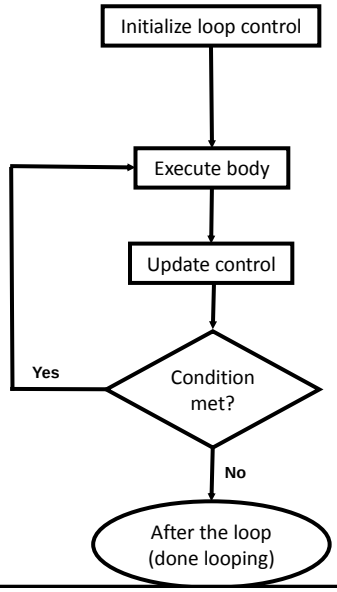
Pre-Test Loops

1. Initialize loop control
2. Check if the repeating condition has been met
   a. If it's been met then go to Step 3)
   b. If it hasn't been met then the loop ends
3. Execute the body of the loop (the part to be repeated)
4. Update the loop control
5. Go to step 2



Post-Test Loops (Not Implemented In Python)

1. Initialize loop control (sometimes not needed because initialization occurs when the control is updated)
2. Execute the body of the loop (the part to be repeated)
3. Update the loop control
4. Check if the repetition condition has been met
   a. If the condition has been met then go through the loop again (go to Step 2)
   b. If the condition hasn't been met then the loop ends.



Pre-Test Loops In Python

1. While
2. For

Characteristics:

1. The stopping condition is checked *before* the body executes.
2. These types of loops execute zero or more times.

Post-Loops In Python

- Note: this type of looping construct has not been implemented with this language.
- But many other languages do implement post test loops.

Characteristics:

- The stopping condition is checked *after* the body executes.
- These types of loops execute one or more times.

The `While` Loop

- This type of loop can be used if it's *not known* in advance how many times that the loop will repeat (most powerful type of loop, any other type of loop can be simulated with a while loop).
  - It can repeat so long as some arbitrary condition holds true.
- **Format:**

(Simple condition)

```
while (Boolean expression):
      body
```

(Compound condition)

```
while (Boolean expression) Boolean operator (Boolean expression):
      body
```

The `While` Loop (2)

- **Program name:** `while1.py`

```
i = 1
while (i <= 3):
      print ("i =", i)
      i = i + 1
print ("Done!")
```

1) Initialize control
2) Check condition
3) Execute body
4) Update control

The `While` Loop (2)

- **Program name:** `while1.py`

```
i = 1
while (i <= 3):
     print ("i =", i)
     i = i + 1
print ("Done!")
```

Tracing The `While` Loop

Execution	**Variable**
>python while1.py	i

Countdown Loop

- **Program name:** while2.py

```
i = 3
while (i >= 1):
     print("i =", i)
     i = i - 1
print("Done!")
```

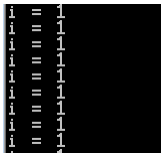
Tracing The Count Down `While` Loop

Execution	**Variable**
>python while2.py	i

Common Mistakes: While Loops

- Forgetting to include the basic parts of a loop.
  - Updating the control

```
i = 1
while (i <= 4):
    print ("i =", i)
```



James Tam

Practice Exercise

- The following program that prompts for and displays the user's age.
- Modifications:
  - As long as the user enters a negative age the program will continue prompting for age.
  - After a valid age has been entered then stop the prompts and display the age.

```
age = int(input("Age: "))
print(age)
```

```
Age: -1
Age cannot be negative
Age: -123
Age cannot be negative
Age: 27
You are 27 years young :P
```

James Tam

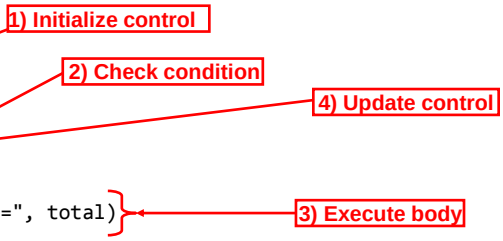
The For Loop

- In Python a `for`-loop is used to step through a sequence e.g., count through a series of numbers or step through the lines in a file.
- **Syntax:**

```
for <name of loop control> in <something that can be iterated>:
    body
```

- **Program name:** `for1.py`

```
total = 0
for i in range (1, 4, 1):
  total = total + i
  print ("i=", i, "\ttotal=", total)
print ("Done!“)
```



The For Loop

- In Python a `for`-loop is used to step through a sequence
- **Syntax:**

```
for <name of loop control> in <something that can be iterated>:
    body
```

- **Program name:** `for1.py`

```
i = 0
total = 0
for i in range (1, 4, 1):
  total = total + i
  print ("i=", i, "\ttotal=", total)
print ("Done!")
```

Tracing The First For Loop Example

Execution

```
>python for1.py
```

Variables

i total

Counting Down With A For Loop

- **Program name:** for2.py

```
i = 0
total = 0
for i in range (3, 0, -1):
    total = total + i
    print ("i = ", i, "\t total = ", total)
print ("Done!")
```

Tracing The Second For Loop Example

Execution

```
>python for2.py
```

Variables

```
i          total
```

Erroneous For Loops

- The logic of the loop is such that the end condition has already been reached with the start condition.
- **Example:** `for_error.py`

```
for i in range (5, 0, 1):
       total = total + i
       print ("i = ", i, "\t total = ", total)
print ("Done!")
```

```
[csc loops 18 ]> python for_error.py
Done!
```

Loop Increments Need Not Be Limited To One

- **While:** `while_increment5.py`

```
i = 0
while (i <= 100):
    print ("i =", i)
    i = i + 5
print ("Done!")
```

- **For:** `for_increment5.py`

```
for i in range (0, 105, 5):
    print ("i =", i)
print ("Done!")
```

```
i = 0
i = 5
i = 10
i = 15
i = 20
i = 25
i = 30
i = 35
i = 40
i = 45
i = 50
i = 55
i = 60
i = 65
i = 70
i = 75
i = 80
i = 85
i = 90
i = 95
i = 100
Done!
```

Sentinel Controlled Loops

- The stopping condition for the loop occurs when the 'sentinel' value is reached.
- **Program name**: `sum.py`

```
total = 0
temp = 0
while (temp >= 0):
     temp = input ("Enter a non-negative integer (negative to end
       series):")
     temp = int(temp)
     if (temp >= 0):
         total = total + temp

print ("Sum total of the series:", total)
```

```
Enter a positive integer (negative to end series):1
Enter a positive integer (negative to end series):2
Enter a positive integer (negative to end series):3
Enter a positive integer (negative to end series):-1
Sum total of the series: 6
```

Q: What if the user just entered a single negative number?

Sentinel Controlled Loops (2)

- Sentinel controlled loops are frequently used in conjunction with the error checking of input.
- Example (sentinel value is one of the valid menu selections, repeat while selection is not one of these selections)

```
selection = " "
while selection not in ("a", "A", "r",  "R", "m", "M", "q", "Q"):
    print ("Menu options")
    print ("(a)dd a new player to the game")
    print ("(r)emove a player from the game")
    print ("(m)odify player")
    print ("(q)uit game")
    selection = input ("Enter your selection: ")
    if selection not in ("a", "A", "r",  "R", "m", "M", "q", "Q"):
        print ("Please enter one of 'a', 'r', 'm' or 'q' ")
```

Recap: What Looping Constructs Are Available In Python/When To Use Them

Construct	When To Use
Pre-test loops	You want the stopping condition to be checked before the loop body is executed (typically used when you want a loop to execute zero or more times).
• `While`	• The most powerful looping construct: you can write a 'while-do' loop to mimic the behavior of any other type of loop. In general it should be used when you want a pre-test loop which can be used for most any arbitrary stopping condition e.g., execute the loop as long as the user doesn't enter a negative number.
• `For`	• In Python it can be used to step through some sequence
Post-test: None in Python	You want to execute the body of the loop before checking the stopping condition (typically used to ensure that the body of the loop will execute at least once). The logic can be simulated with a `while` loop.

The Break Instruction

Q: What if the user just typed 'abc' and hit enter?

- It is used to terminate the repetition of a loop which is separate from the main Boolean expression (it's another, separate Boolean expression).
- **General structure**:

```
for (Condition 1):
    if (Condition 2):
       break
```

```
while (Condition 1):
    if (Condition 2):
        break
```

- Specific example (mostly for illustration purposes at this point): `break.py`

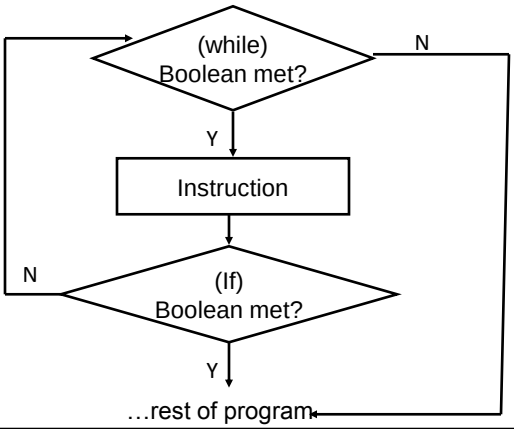
```
str = input("Enter a series of lower case alphabetic characters: ")
for temp in str:
    if (temp < 'a') or (temp > 'z'):
        break
    print(temp)
print("Done")
```

```
Enter a series of lower case alphabetic characters: abcD
a
b
c
Done
```

James Tam

The Break Should Be Rarely Used

- Adding an extra exit point in a loop (aside from the Boolean expression in the while loop) may make it harder to trace execution (leads to 'spaghetti' programming).



JT: While adding a single break may not always result in 'spaghetti' it's the beginning of a bad habit that may result in difficult to trace programs

James Tam

An Alternate To Using A 'Break'

- Instead of an 'if' and 'break' inside the body of the loop

```
while (BE1):
    if (BE2):
       break
```

- Add the second Boolean expression as part of the loop's main Boolean expression

```
while (BE1) and not (BE2):
```

James Tam

Another Alternative To Using A 'Break'

- If the Boolean expressions become too complex consider using a 'flag'

```
flag = true
while (flag == true):
    if (BE1):
        flag == false
    if (BE2)
        flag == false
    # ETC.
```

- Both of these approaches still provide the advantage of a single exit point from the loop.

James Tam

Nested Loops

- One loop executes inside of another loop(s).
- **Example structure**:

```
Outer loop (runs n times)
    Inner loop (runs m times)
        Body of inner loop (runs n x m times)
```

- Program name: `nested.py`

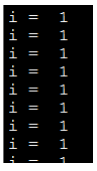
```
i = 1
while (i <= 2):
    j = 1
    while (j <= 3):
        print("i = ", i, " j = ", j)
        j = j + 1
    i = i + 1
print("Done!")
```

```
i =  1  j =  1
i =  1  j =  2
i =  1  j =  3
i =  2  j =  1
i =  2  j =  2
i =  2  j =  3
Done!
```

Infinite Loops

- Infinite loops never end (the stopping condition is never met).
- They can be caused by logical errors:
  - The loop control is never updated (Example 1 – below).
  - The updating of the loop control never brings it closer to the stopping condition (Example 2 – next slide).
- **Example 1**: `infinite1.py`

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



To stop a program with an infinite loop in Unix simultaneously press the <ctrl> and the <c> keys

Infinite Loops (2)

- **Example 2**: `infinite2.py`

```
i = 10
while (i > 0):
    print ("i = ",  i)
    i = i + 1
print("Done!")
```

```
i =  14477
i =  14478
i =  14479
i =  14480
i =  14481
i =  14482
i =  14483
```

To stop a program with an infinite loop in Unix simultaneously press the <ctrl> and the <c> keys

Testing Loops

- Make sure that the loop executes the proper number of times.
- Test conditions:
  1) Loop does not run
  2) Loop runs exactly once
  3) Loop runs exactly 'n' times

Testing Loops: An Example

```
sum = 0
i = 1
last = 0

last = int(input ("Enter the last number in the sequence to sum : "))
while (i <= last):
    sum = sum + i
    print ("i = ", i)
    i = i + 1

print ("sum =", sum)
```

Extra Practice

- Write a loop that will continue repeating if the user enters a value that is negative.
- Write a program that will prompt the user for number and an exponent. Using a loop the program will calculate the value of the number raised to the exponent.
  - To keep it simple you can limit the program to non-negative exponents.

James Tam

Problem Solving: Using Loops For A More Complex Problem

- Write a program that will prompt the user for the birth month and the day of birth.
- The birth month must be a value from 1 – 12.
- The day of birth must be a number that is one or greater while the maximum value will be determined by the maximum days in a particular month.

  – *Thirty days hath September,*
  April, June, and November:
  All the rest have thirty-one,
  Except for February,
  Which hath twenty-eight days clear,
  And twenty-nine in each leap year.
    - *[JT's note: for this example you can assume that a day of birth of 29 is always valid for February]*

James Tam

Problem Solving: Using Loops For A More Complex Problem (2)

- The program will prompt first for the month and repeat the prompt as long as the value is not within the valid range.
- Next the program will prompt for day of birth and repeatedly prompt for a value so long as day is valid for the particular month (see previous slide).
- After receiving a valid month and day the program will display the month of birth and the day of birth.

James Tam

Pseudo Code

- A high level solution or algorithm that is not specified in a programming language.
- Instead English-like statements are used.
  - "A high-level description of the actions of a program or algorithm, using a mixture of English and informal programming language syntax" – Python for Everyone (Horstmann, Necaise)
- Benefits: it allows the programmer to focus on the solution without spending a lot time worrying about details such as syntax.
- When the pseudo code solution has been created then those details can be handled separately when the solution is translated into an actual program.
- For difficult problems it may be beneficial to break the problem solving process into these steps.

James Tam

A Pseudo Code Solution

```
While (month is not between 1 and 12) do
    Prompt user for month
If (month is one with 31 days) then
    while (month is not between 1 and 31) do
        Prompt user for day
If (month is one with 30 days) then
    while (month is not between 1 and 30) do
        Prompt user for day
If (month has 29 days) then
    while (month is not between 1 and 29) do
        Prompt user for day
Show month and day of birth
```

James Tam

Program Code Solution

- **Program name**: `calendar.py`

```
# Month separate: once value is entered it cannot be changed
month = -1
while (month < 1) or (month > 12):
    month = int(input("Enter value for month (1 - 12): "))
    if (month < JAN) or (month > DEC):
        print("Month must be a value from 1 - 12")

# Jan, Mar, May, July, Aug, Oct, Dec
if (month in (1,3,5,7,8,10,12)):
    day = -1
    while (day < 1) or (day > 31):
        day = int(input("Enter value for day (1 - 31): "))
        if (day < 1) or (day > 31):
            print("Month must be a value from 1 - 31")
```

```
Enter value for month (1 - 12): 0
Month must be a value from 1 - 12
Enter value for month (1 - 12): 13
Month must be a value from 1 - 12
Enter value for month (1 - 12): 2
```

James Tam

Program Code Solution (2)

```
# April, June, Sept, Nov
elif (month in (4,6,9,11)):
    day = -1
    while (day < 1) or (day > 30):
        day = int(input("Enter value for day (1 - 30): "))
        if (day < 1) or (day > 30):
            print("Month must be a value from 1 - 30")
# February
elif (month == 2):
    day = -1
    while (day < 1) or (day > 29):
        day = int(input("Enter value for day (1 - 29): "))
        if (day < 1) or (day > 29):
            print("Month must be a value from 1 - 29")

print()
print("Birth information")
print("Birth month: %d" %month)
print("Day of birth: %d" %day)
```

```
Enter value for day (1 - 29): 30
Month must be a value from 1 - 29
Enter value for day (1 - 29): 29
```

```
Birth information
Birth month: 2
Day of birth: 29
```

James Tam

Extra Practice

- Modify the program so that the user can select a different month after a day has been entered.
  - It will still prompt for and error check the month
  - Next it will prompt for and error check the day
  - If the combination is valid then the program will simply display the month/day.
  - If the combination is not valid then the program will re-prompt for the month (repeat the process until a combination is valid).
- How does this change in the program requirements change the solution?
  - Try modifying the pseudo code solution before producing a solution in Python.

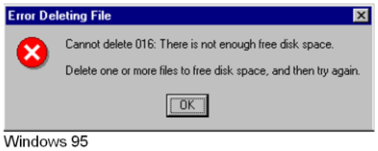
James Tam

User-Friendly Software

- In today's world it's not just sufficient to create software that has implemented a given set of operations.
- If the person using the system cannot understand it or has troubles using common functions then the software or technology is useless.
- Reference course: If you're interested in more information:
  - http://pages.cpsc.ucalgary.ca/~tamj/2008/481W/index.html

Not So Friendly Examples

Do I have any choice in this? [AXE a hex editor]



Windows 95

Uhhh… I give up on this one [Mac shareware version of RISK]

Some Rules For Designing Software

- (The following list comes from Jakob Nielsen's 10 usability heuristics from the book "*Usability Engineering*"
  1. Minimize the user's memory load
  2. Be consistent
  3. Provide feedback
  4. Provide clearly marked exits
  5. Deal with errors in a helpful and positive manner

1. Minimize The User's Memory Load

- Computers are good at 'remembering' large amounts of information.
- People are not so good remembering things.

1. Minimize The User's Memory Load

- To reduce the memory load of the user:
  - Describe required the input format, show examples of valid input, provide default inputs
- Examples:

Example 1:

Example 2:

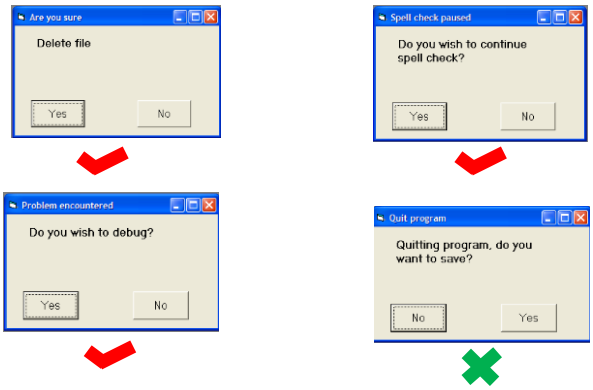
```
[csc loops 25 ]> python hci.py
Enter your birthday <month> <day> <year> e.g., 11 17 1977
Birthday:
```

2. Be Consistent

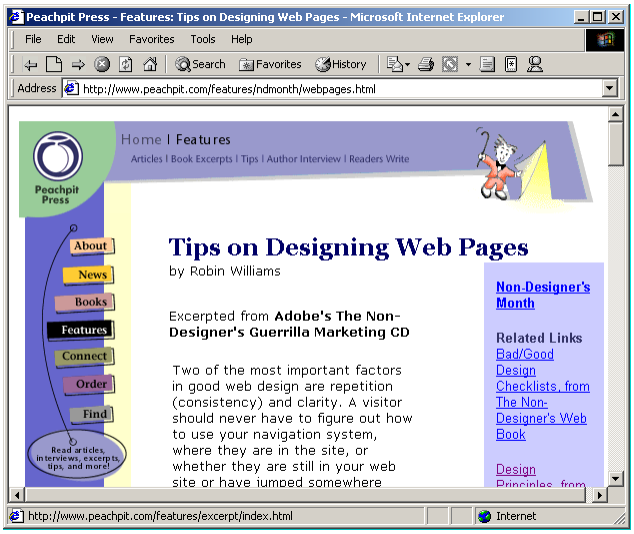
- Consistency of effects
  - Same words, commands, actions will always have the same effect in equivalent situations
  - Makes the system more predictable
  - Reduces memory load
- Consistency of layout
  - Allows experienced users to predict where things should be (matches expectations)

2. Be Consistent

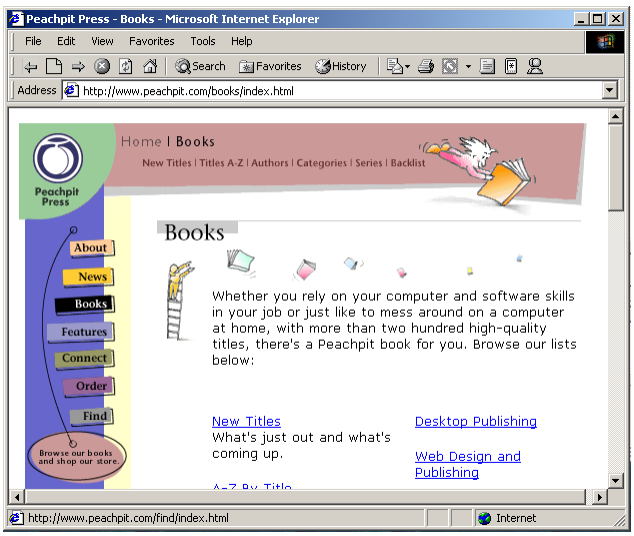
- Consistency of language and graphics
  - Same information/controls in same location on all screens / dialog boxes forms follow boiler plate.
  - Same visual appearance across the system (e.g. widgets).



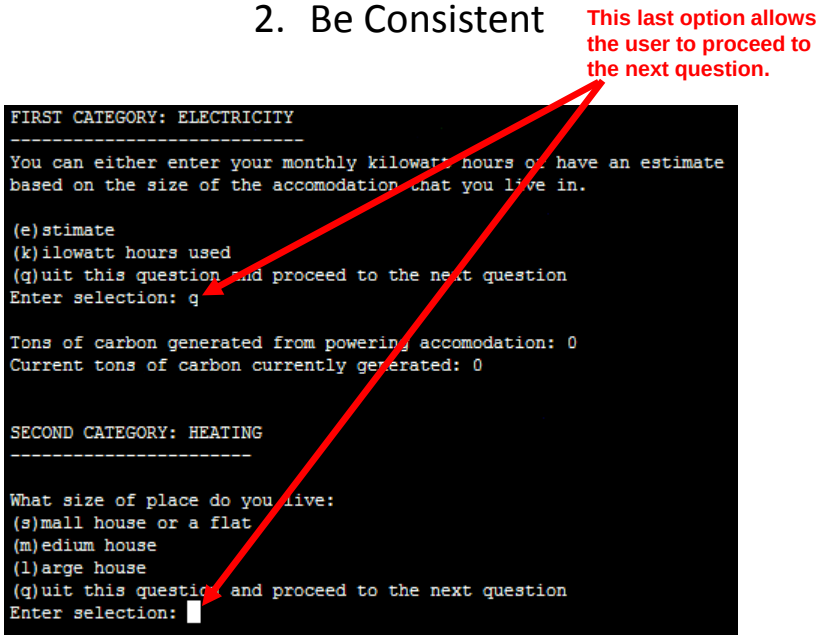
2. Be Consistent



2. Be Consistent



2. Be Consistent

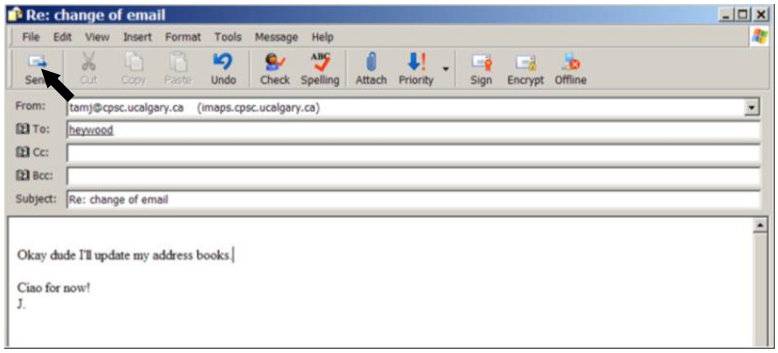


3. Provide Feedback

- Letting the user know:
  - What the program is currently doing: was the last command understood, has it finished with it's current task, what task is it currently working on, how long will the current task take etc.

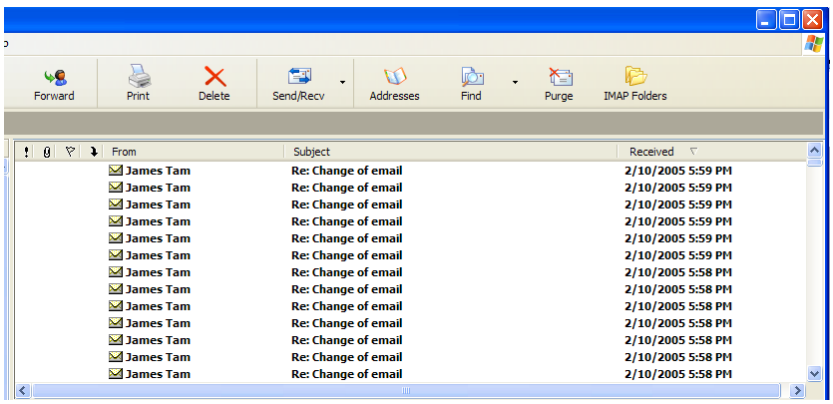
3. Provide Feedback

- What is the program doing?



3. Provide Feedback

- The rather unfortunate effect on the (poor) recipient.



3. Provide Feedback

- In terms of this course, feedback is appropriate for instructions that may not successfully execute
  - what the program is doing (e.g., opening a file),
  - what errors may have occurred (e.g., could not open file),
  - and why (e.g., file "`input.txt`" could not be found)
- ...it's not hard to do and not only provides useful updates with the state of the program ("Is the program almost finished yet?") but also some clues as to how to avoid the error (e.g., make sure that the input file is in the specified directory).
- At this point your program should at least be able to provide some rudimentary feedback
  - E.g., if a negative value is entered for age then the program can remind the user what is a valid value (the valid value should likely be shown to the user as he or she enters the value):

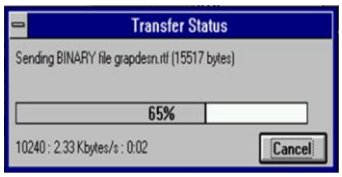
```
age = int(input ("Enter age (0 – 114): "))
```

4. Provide Clearly Marked Exits

- This should obviously mean that quitting the program should be self-evident (although this is not always the case with all programs!).
- In a more subtle fashion it refers to providing the user the ability to reverse or take back past actions (e.g., the person was just experimenting with the program so it shouldn't be 'locked' into mode that is difficult to exit).
- Users should also be able to terminate lengthy operations as needed.

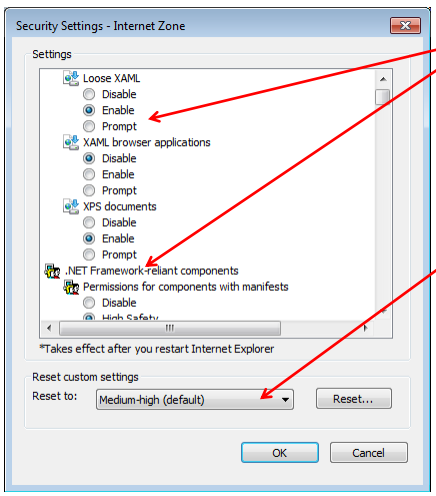
4. Provide Clearly Marked Exits

- This doesn't just mean providing an exit from the program but the ability to 'exit' (take back) the current action.
  - Universal Undo/Redo
    - e.g., <Ctrl>-<Z> and <Ctrl>-<Y>
  - Progress indicator & Interrupt
  - Length operations



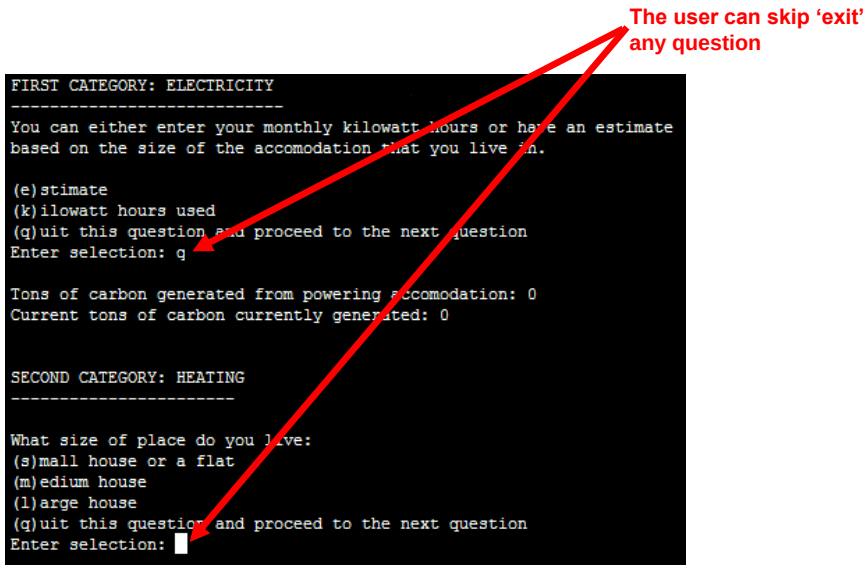
4. Provide Clearly Marked Exits

- Restoring defaults
  - Getting back original settings



- **What option did I change?**
- **What was the original setting?**
- **Allows for defaults to be quickly restored**

4. Provide Clearly Marked Exits



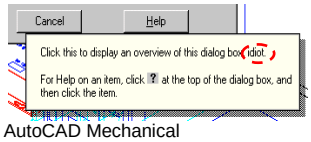
5. Deal With Errors In A Helpful And Positive Manner

- (JT: with this the heuristic it states exactly what should be done).

Rules Of Thumb For Error Messages

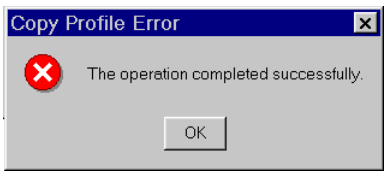
1. Polite and non-intimidating
   - Don't make people feel stupid
     - `Try again, bonehead!` ← No
2. Understandable
   - Error 25 ← Not
3. Specific
   - Cannot open this document
   - Cannot open "chapter 5" because the application "Microsoft Word" is not on your system
4. Helpful
   - Cannot open "chapter 5" because the application "Microsoft Word" is not on your system. Open it with "WordPad" instead?

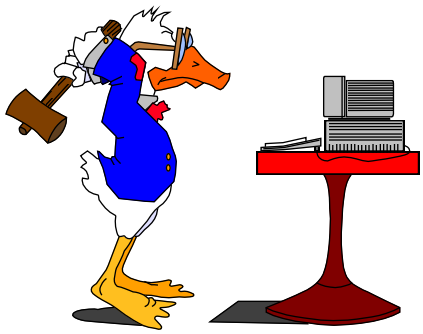
So obvious it could never happen?



AutoCAD Mechanical

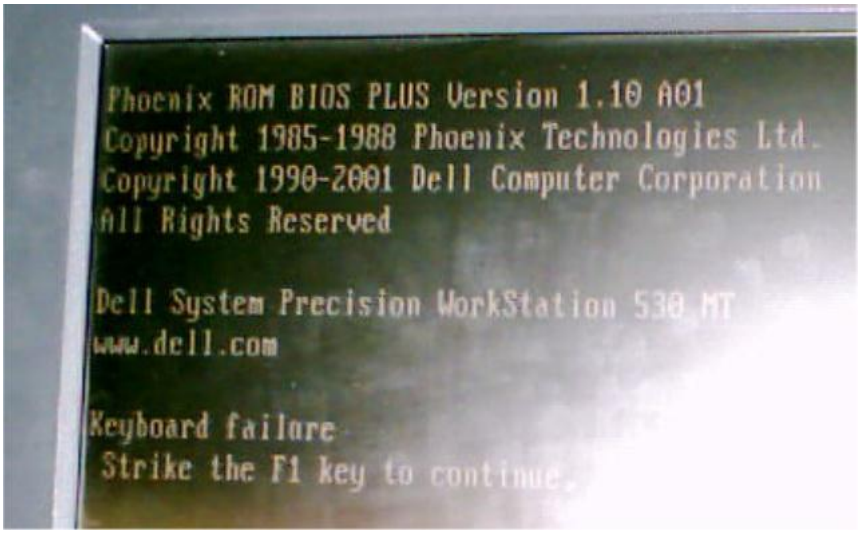
Examples Of Bad Error Messages





“HIT ANY KEY TO CONTINUE”

I Think I'd Rather Deal With The Any Key!!!



Picture courtesy of James Tam: An error message from a Dell desktop computer

After This Section You Should Now Know

- When and why are loops used in computer programs
- What is the difference between pre-test loops and post-test loops
- How to trace the execution of pre-test loops
- How to properly write the code for a loop in a program
- What are nested loops and how do you trace their execution
- How to test loops
- Some rules of thumb for interaction design
  1. Minimize the user's memory load
  2. Be consistent
  3. Provide feedback
  4. Provide clearly marked exits
  5. Deal with errors in a helpful and positive manner