CHAMP

http://www.boeing.com/Features/2012/10/bds_champ_10_22_12.html

- Questions from homework?

Post-Test Loop

- Write a program to get a nonnegative number from user
- If user types an incorrect input, the program asks for another value
- Process continues until a valid value has been entered
- input validation

Post-Test Loop

- When condition test comes after body of the loop it's called a post-test loop
- A post-test loop always executes the body of the code at least once
- Python doesn't have a built-in statement
- Can do it with a `while` loop

Post-Test Loop

- Seed the loop condition so we're guaranteed to execute the loop once

```
number = -1
while number < 0:
    number = eval(input("Enter a positive number: "))
```

- By setting `number` to –1, we force the loop body to execute at least once

Post-Test Loop -- Break

- Some programmers prefer to simulate a post-test loop by using `break`
- Executing `break` causes Python to immediately exit the loop
- `break` is sometimes used to exit what looks like an infinite loop

Post-Test Loop -- Break

The same algorithm implemented with a `break`:

```
while True:
      number = eval(input("Enter a positive number: "))
      if x >= 0: break # Exit loop if number is valid
```

- A while loop continues as long as the expression evaluates to true. Since `True` *always* evaluates to true, it looks like an infinite loop!

Post-Test Loop -- Break

- When the value of *x* is nonnegative, the `break` statement executes, which terminates the loop
- If the body of an `if` is only one line long, you can place it right after the `:`

Post-Test Loop -- Break

- In the `while` loop version, this is awkward:

```
number = -1
while number < 0:
    number = eval(input("Enter a positive number: "))
    if number < 0:
        print("The number you entered was not positive")
```

- Validity check in two places….

Post-Test Loop -- Break

- Adding the warning to the `break` version only adds `else` statement:

```
while True:
    number = eval(input("Enter a positive number: "))
    if x >= 0:
        break # Exit loop if number is valid
    else:
        print("The number you entered was not positive.")
```

Loop and a Half

- Some programmers prefer the following approach:

```
while True:
    number = eval(input("Enter a positive number: "))
    if x >= 0: break # Loop exit
    print("The number you entered was not positive")
```

- Here the loop exit is in the middle of the loop body. This is what we mean by a *loop and a half*.

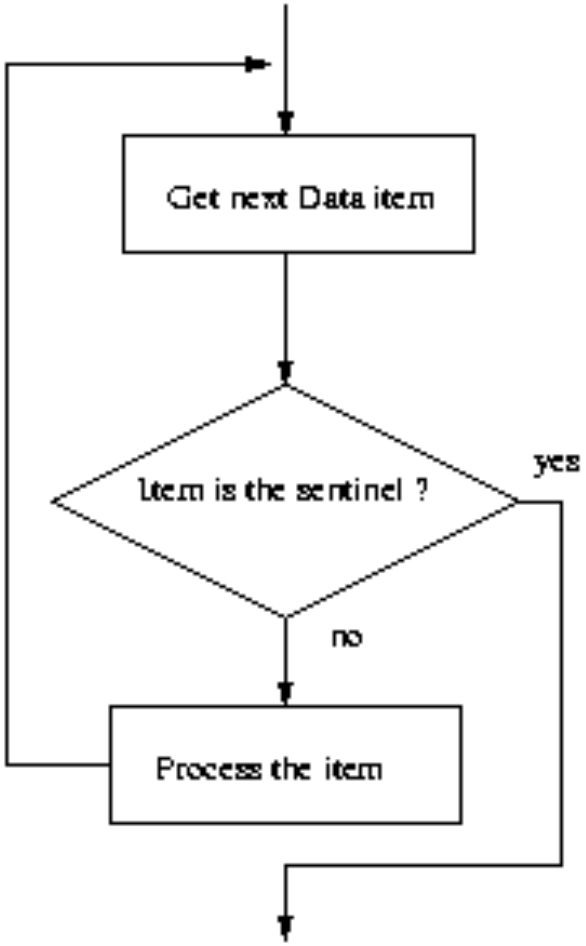
Loop and a Half

- The loop and a half is an elegant way to avoid the priming read in a sentinel loop

```
while True:
    get next data item
    if the item is the sentinel: break
    process the item
```

- This method is faithful to the idea of the sentinel loop, the sentinel value is not processed!

Loop and a Half



Loop and a Half

- Use of break is mostly a matter of style and taste
- Avoid using break often within loops, because the logic of a loop is hard to follow when there are multiple exits

Boolean Expressions as Decisions

- Boolean expressions can be used as control structures themselves
- Suppose you're writing a program that keeps going as long as the user enters a response that starts with 'y' (like our interactive loop)
- One way you could do it:

```
while response[0] == "y" or
      response[0] == "Y"   :
```

Boolean Expressions as Decisions

- Be careful! You can't take shortcuts:
  ```
  while response[0] == "y" or "Y":
  ```
- Why doesn't this work?
- Python has a `bool` type that internally uses 1 and 0 to represent `True` and `False`, respectively
- The Python condition operators, like `==`, always evaluate to a value of type `bool`

Boolean Expressions as Decisions

- However, Python will let you evaluate any built-in data type as a Boolean
- For numbers (int, float, and long ints), zero is considered `False`, anything else is considered `True`

Boolean Expressions as Decisions

```
>>> bool(0)
False
>>> bool(1)
True
>>> bool(32)
True
>>> bool("Hello")
True
>>> bool("")
False
>>> bool([1,2,3])
True
>>> bool([])
False
```

Boolean Expressions as Decisions

- Empty sequence is interpreted as `False`
- Non-empty sequence is `True`
- The Boolean operators have operational definitions that make them useful for other purposes

Boolean Expressions as Decisions

Operator	Operational definition
x `and` *y*	If *x* is false, return *x*. Otherwise, return *y*.
x `or` *y*	If *x* is true, return *x*. Otherwise, return *y*.
`not` *x*	If *x* is false, return `True`. Otherwise, return `False`.

Boolean Expressions as Decisions

- Consider *x* `and` *y*. In order for this to be true, both *x* and *y* must be true
- As soon as one is found to be false, we know the expression as a whole is false. Don't need to finish evaluating the expression
- If *x* is false, Python should return a false result, namely *x*

Boolean Expressions as Decisions

- If *x* is true, then whether the expression as a whole is true or false depends on *y*
- By returning *y*, if *y* is true, then true is returned. If *y* is false, then false is returned

Boolean Expressions as Decisions

- These definitions show that Python's Booleans are short-circuit operators, meaning that a true or false is returned as soon as the result is known
- In an `and` where the first expression is false and in an `or`, where the first expression is true, Python will not evaluate the second expression

Boolean Expressions as Decisions

- `response[0] == "y" or "Y“`
- The Boolean operator is combining two operations.
- Here's an equivalent expression:
  `(response[0] == "y") or ("Y")`
- By the operational description of `or`, this expression returns either `True`, if response[0] equals "y", or "Y", both of which are interpreted by Python as true

Boolean Expressions as Decisions

- Sometimes we write programs that prompt for information but offer a default value obtained by simply pressing `<Enter>`
- Since the string used by `ans` can be treated as a Boolean, the code can be further simplified

Boolean Expressions as Decisions

- ```
  ans = input("What flavor fo you want [vanilla]: ")
  if ans:
      flavor = ans
  else:
      flavor = "vanilla"
  ```
- If the user just hits <Enter>, `ans` will be an empty string, which Python interprets as false.
  ```

Boolean Expressions as Decisions

- We can code this even more succinctly!

```
ans = input("What flavor fo you want [vanilla]: ")
flavor = ans or "vanilla“
```

- Remember, any non-empty answer is interpreted as `True`
- Could be boiled down into one line…

```
flavor = input("What flavor do you want
         [vanilla]:” ) or "vanilla"
```

Boolean Expressions as Decisions

- Again, if you understand this method, feel free to use it. Just make sure that if your code is tricky, that it's well documented!

- http://cheezburger.com/44047361

- Lab 8: Random numbers

```
from random import *

randrange(0,100,10)
random()
```

PseudoRandom Numbers

- Many simulations require events to occur with a certain likelihood
  - These sorts of simulations are called Monte Carlo simulations because results depend on "chance"

- Chaos program from chapter 1:
  - Apparent randomness of the result came from repeatedly applying a function to generate a sequence of numbers
  - A similar approach is used to generate random (technically *pseudorandom*) numbers

PseudoRandom Numbers

- A pseudorandom number generator works by starting with a *seed* value. This value is given to a function to produce a "random" number
- The next time a random number is required, current value is input to function to produce new number

PseudoRandom Numbers

- This sequence of numbers appears to be random, but if you start the process over again with the same seed number, you'll get the same sequence of "random" numbers
- Python provides a library module that contains a number of functions for working with pseudorandom numbers

PseudoRandom Numbers

- These functions derive an initial seed value from the computer's date and time when the module is loaded, so each time a program is run a different sequence of random numbers is produced

PseudoRandom Numbers

- `randrange` function is used to select a pseudorandom int from a given range
- Syntax is similar to that of `range` command
- `randrange(1,6)`
  - returns some number from `[1,2,3,4,5]` and
- `randrange(5,105,5)`
  - returns a multiple of 5 between 5 and 100, inclusive

PseudoRandom Numbers

- Each call to `randrange` generates a new pseudorandom int.

```
>>> from random import randrange
>>> randrange(1,6)
5
>>> randrange(1,6)
3
>>> randrange(1,6)
2
>>> randrange(1,6)
5
>>> randrange(1,6)
5
>>> randrange(1,6)
5
>>> randrange(1,6)
4
```

PseudoRandom Numbers

- In the example, 5 comes up over half the time, demonstrating probabilistic nature of random numbers
- Over time, this function will produce a uniform distribution: all values will appear an approximately equal number of times

PseudoRandom Numbers

- `random` function is used to generate pseudorandom floating point values
- Takes no parameters
- Returns values uniformly distributed between 0 and 1 (including 0 but excluding 1)

PseudoRandom Numbers

```
>>> from random import random
>>> random()
0.79432800912898816
>>> random()
0.00049858619405451776
>>> random()
0.1341231400816878
>>> random()
0.9872454535361653
>>> random()
0.21429424175032197
>>> random()
0.2390358371212714l
>>> random()
0.72918328843408919
```

Simulating Racquetball

- *Simulation* can solve real-world problems by modeling real-world processes
- Provide otherwise unobtainable information
- Computer simulation is used to predict the weather, design aircraft, create special effects for movies, etc.
- *http://www.youtube.com/watch?v=4jf8W2rwbXw*