Introduction To Python

Week 3: Control Structures

Dr. Jim Lupo
Asst Dir Computational Enablement
LSU Center for Computation & Technology
jalupo@cct.lsu.edu

Resources

- Noteworthy additional reference:

  "Think Python: How to Think Like A Computer Scientist", Version 2.0.15, Mar 2015 by Allen Downey

  `http://reu.cct.lsu.edu/documents/Python_Course/thinkpython.pdf`

On To Control Structures

- Switch from data to control.
- Express desired actions based on conditions.

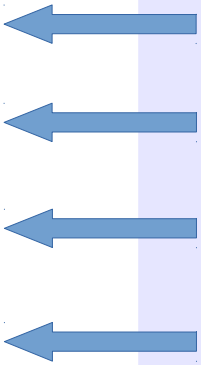
The IF Statement

- Basic decision: Do something is True
- Optionally: Do something else if False.
- Allow multiple tests.

If **RED**, then ***stop***, else if **GREEN**, then ***go***, else if **YELLOW**, ***go really fast***.

Do Something If True

```
>>> x = True
>>> if x :
        print 'Yeah!'

Yeah!
>>>
```



IF statement

Indented statement block

End of indentation

Expected result for True

Replace **x** with any *expression* that returns a boolean value.

Code structure based on indendation!

Full-Blown IF

```
if x :
    print 'Yeah'
    more statements in block
elif y :
    print 'Booyha!'
    if z :
        more statements in block
elif <other elif sections>
else :
    print 'Oh oh!'
    more statements in block
```

Note the indentation.

Indent level: 1 2

Nested IF Statements

```
x = False
y = True
print 'Nested IF'
if x :
    print 'Yeah'
    if y :
        print 'Booyha!'
    else:
        print 'Oh, oh!'
else:
    print 'Not good!'
print 'Moving on.'
```

Create a file nested.py. Play with changes to **x** and **y**. Will revisit this program later, so save it.

Programming Style Comment

- This is legal: **if x:**
- So is this: **if x :**
- Personally tend to prefer more whitespace over dense text.
- Programming style may be specified for you.
- Pick something you like and stick to it. The consistency helps when reading back over older code.

Comparison Operators.

- We saw variables have built in comparison methods.
- In general, use comparison (or relational) *operators*.

==	equality test:	a == 42.
!=	inequality test:	a != 42.
<>	same as !=	a <> 42.
>	greater than test:	a > 42.
>=	greater than or equal to test:	a >= 42.
<	less than test:	a < 42.
<=	less than or equal to test:	a <= 42.

Use Boolean Expressions

- Operator approach

```
if x > 5 :
    print x, 'is greater than 5'
else
    print x, 'is less than or equal to 5'
```

- Comparison method approach

```
if x.__gt__(5) :
    print x, 'is greater than 5'
else
    print x, 'is less than or equal to 5'
```

Brain Teasers

```
x = 'Mary had a little lamb whose fleece' \
    + 'was white as snow.'
```

- Try doing with and without the '****' - line continuation
- What does **x.find('z')** produce?
- What does **x.index('z')** do?
- How would you determine the index of some letter in **x**?

A Solution

```
x = 'Mary had a little lamb whose fleece' \
     + 'was white as snow.'
if x.find('z') >= 0 :
    i = x.index('z')
    print 'There is a z in x at index', i
else :
    print 'Sorry, no z in x!'
```

Beyond IF

- Much of the power of programming comes from being able to do different things based on information provided.
- IF statements allow ***branching***.
- Other statements allow repetition, ***looping***.

The FOR Loop

- The FOR loop is the simplest way to do iteration - repeat statement block with a different value some specified number of times (i.e. once per list member). The syntax looks like:

```
for x in iterable :
    statement block
```

- *Iterable* is just a fancy way of saying it wants the *equivalent* of a list of items. To wit:

```
for x in ['a','b','c'] :
    statement block
```

Simple Examples

```
x = 'Mary had a little lamb whose fleece' \
    + 'was white as snow.'

# Print out each character:

for c in x :
    print c

# Print out each word:
for w in x.split() :
    print w
```

Counting Loop with range()

- If you need to run something 10,000 times, you don't want to have to manually create a list with 10,000 entries.
- Let **range()** create it for you!

```
>>> range(3)
[0, 1, 2]
>>> range(6,10)
[6, 7, 8, 9]
>>> range(100,200,10)
[100, 110, 120, 130, 140, 150, 160, 170, 180, 190]
>>>
```

Using Range in a For Statement

- What does this do?

```
for i in range(100) :
    if i%2 :
        print i**2, i**3, i**4
```

Dictionary Iteration

- Dictionaries provide an iterable method!

```
>>> example = {'a':1,'b':2,'c':42}
>>> for k, v in example.iteritems() :
        print k, v

a 1
c 42
b 2
>>>
```

- Or do it manually:

```
>>> for k in example.keys() :
        print k, example[k]

a 1
c 42
b 2
>>>
```

Did You Note IDLE Help?

```
>>>range(
```

range(stop) -> list of integer
range(start,stop[,step]) -> list of integers

range(stop) - list from 0 to stop-1
range(start,stop) - list from start to stop-1
range(start,stop,step) - list from start to stop-1 by step

The notation **[,step]** means the argument is optional.
It does not imply a list.

Correct or Not?

1) range(0,12,2) => [0,2,4,6,8,10]
2) range(7,3,-1) => [7,6,5,4,3]
3) range(-3) => [0,-1,-2]

Easy To Test

```
>>> range(0,12,2)
[0, 2, 4, 6, 8, 10]
>>> range(7,3,-1)
[7, 6, 5, 4]
>>> range(-3)
[]
```

List of Capital Letters

- The ASCII table gives the binary value of the printable keyboard letters.
- A-Z just happen to be 65-90
- The function chr(i) will return the ASCII letter that has value i.
- How would you create a list of capital letters?

The Alphabet: Possible Solution

Code:

```
alphabet=[]
for i in range(65,91):
    alphabet.append(chr(i))
print alphabet
```

Output:

```
['A', 'B', 'C', . . . , 'Z']
```

Sorted Dictionary

- Recall that entries in a dictionary are in some internally determined order?
- How would you produce alphabetical output for a word frequency table with entries consisting of: 'word':integer
- Call the dictionary 'counts'.

Sorted Dictionary: A Solution

Code:

```
counts = {'apple':1, 'ants':3,'zebras':6,'swans':42}
print counts
keys = counts.keys()
keys.sort()
for w in keys :
    print w, counts[w]
```

Output:

```
{'swans': 42, 'apple': 1, 'ants': 3, 'zebras': 6}
ants 3
apple 1
swans 42
zebras 6
```

The WHILE Loop

- What if you're not sure how many times to do something?
- Use the WHILE loop to keep going as long as some *expression* is True.

```
while expression :
    statement block
```

Error Convergence

- A great example is repeating (iterating) a calculation until the error estimate is less than some value:

```
err = 1.0
maxerr = 1.0e-7
new = starting_value

while err > maxerr :
    last = new
    compute new value
    err = (last - new)/new

. . . more stuff . . .
```

Geeky Example: Machine Epsilon

- Machine **ε** estimates the smallest number such that:

$$1.0 - \varepsilon \; != \; 1.0$$

- Binary numbers aren't perfect.
- This issue is called: *floating point underflow*

```
x = 1.0
i = 1
while (1.0 - x/2.0) != 1.0 :
    i += 1
    x /= 2.0

print i, x
```

Assignment Operators?

- There are assignment operators that provide short hand statements:

iterationCount = iterationCount + 1

or

iterationCount += 1

- They are:

+= *= /= -= **= %= //=

Problem

- Given a list of fruit names, specify 3 different ways of listing every other entry.
- Use: apple, pear, peach, pineapple, raspberry, strawberry, blackberry, lime, lemon, orange, mango, banana

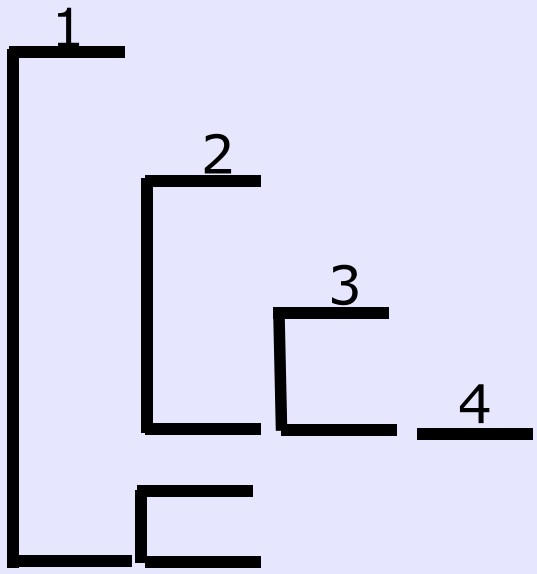
Possible Solution

```
fruits = ['apple', 'pear', 'peach',
          'pineapple', 'raspberry',
          'strawberry', 'blackberry',
          'lime', 'lemon', 'orange',
          'mango', 'banana']

print 'For Loop'

for i in range(len(fruits)) :
    if i%2 == 0:
        print fruits[i]

print 'List Comprehension'

for n in fruits[0::2] :
    print n

print 'List While Loop'
i = 0
while i < len(fruits) :
    if i%2 == 0:
        print fruits[i]
    i += 1
```

```
>>>
For Loop
apple
peach
raspberry
blackberry
lemon
mango
List Comprehension
apple
peach
raspberry
blackberry
lemon
mango
List While Loop
apple
peach
raspberry
blackberry
lemon
mango
>>>
```

Nested Control Structures

- Arbitrary control structures within the statement blocks of other control structures!
- Indentation is important!




```
while true:
    yada..yada
    for x in range(1000) :
        fee_fie_foo
        if foo == 'y' :
            mumble_mumble
            while z > 0.0 :
                zoom_zoom
    for y in range(3) :
        lions_tigers
and still more stuff ...
```

Loop Control Statements

- There are 3 statements that can make life easier when nesting structures:
  - break - immediately leave current structure.
  - continue - immediately start next pass
  - pass - legal do-nothing command

break Command

- Consider a case were you expect 1000 iterations indicates something is going wrong, but are willing to try up to that many:

```
for k in range(1000) :
    something
    something else
    if result == desired :
        break
    still more something
if k == 999 :
    check if really bad
    take action
more statements
```

Infinite Loop

- You may want a server code to run until it receives a 'stop' command:

```
# Set up service, then:
while True :
    cmd = next_command()
    if cmd == 'stop' :
       break
    process command
shut down service
```

continue Command

- The continue command stops processing the current iteration and goes on with the next one:

```
for i in range(100) :
    if i%2 == 0 :
        continue
    print i
```

- Equivalent to:

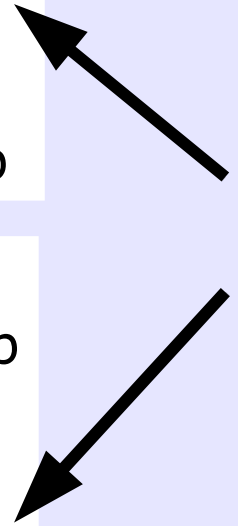
```
for i in range(1,100,2) :
    print i
```

pass Command

- Use any where you need an expression that does nothing.
- For instance: use to reverse sense of a logic test, or limit indenting:

```
if cmd == 'run' :
    do lots of stuff
    all indented
else:
    set up to stop
```

```
if cmd != 'run' :
    set up to stop
else:
    pass
do lots of stuff
not indented.
```



Logically equivalent!

Revisit Nested IF Problem

```
x = False
y = True
print 'Nested IF'
if x :
    print 'Yeah'
    if y :
        print 'Booyha!'
    else:
        print 'Oh, oh!'
else:
    print 'Not good!'
print 'Moving on.'
```

Statement block on next page.

Modify to do all 4 possible pairs of **x, y**?

Different loop setups

1

```
values = [[True,True],[True,False],[False,True],[False,False]]
for x, y in values :
    statement block
```

2

```
for x in [False, True] :
    for y in [False, True] :
        statement block
```

3

```
for i in range(4) :
    if i == 0 :
        x = True; y = True
    elif i == 1 :
        x = True; y = False
    elif i == 2 :
        x = False; y = True
    else :
        x = False; y = False
    statement block
```

Requirements: Score Averaging

- Your task, should you chose to accept is to write a program which allows someone to enter any number of bowling scores (legal score is 0 to 300).
- Report the number entered, the minimum, maximum, and average score.
- Optionally, reject illegal scores.

Program Design

- How will you prompt user?
- What data structure will you use?
- How will user indicate all scores entered?
- How will bad scores be handled?

Bowling Score Program

```
count = 0
sum = 0
max_score = 0
min_score = 300
more = True
while more :
    s = raw_input('Next Score [q to quit]: ')
    if s == 'q' :
        more = False
        break
    if not s.isdigit() :
        print 'Bad Input!'
        continue
    score = float(s)
    if score < 0 or score > 300 :
        print 'Score must be between 0 and 300!'
        continue
    count += 1
    sum += score

print 'Average of ', count, 'scores is ', float(sum)/float(count)
```

Necessary?

A handy string method.