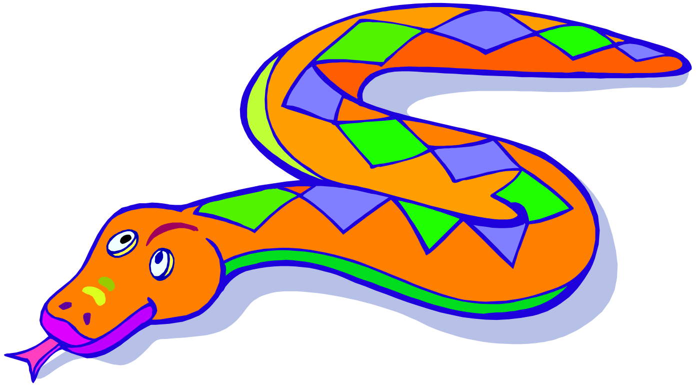
Intro to Python

- Logical expressions
- Control flow
- List comprehensions
- String operations
- Console I/O

Logical Expressions



True and False

- ***True*** **and** ***False*** **are constants in Python.**

- **Other values equivalent to** ***True*** **and** ***False*****:**
  - *False*: zero, *None*, empty container or object
  - *True*: non-zero numbers, non-empty objects

- **Comparison operators: ==, !=, <, <=, etc.**
  - X and Y have same value: `X == Y`
  - Compare with `X is Y` :

    X and Y are two variables that refer to the *identical same object.*

Boolean Logic Expressions

- **You can also combine Boolean expressions.**
    - *True* if a is true and b is True: `a and b`
    - *True* if a is true or b is True: `a or b`
    - *True* if a is False: `not a`

- **Use parentheses as needed to disambiguate complex Boolean expressions.**

Special Properties of *and* and *or*

- **Actually *and* and *or don't* return *True* or *False*.**
- **They return the value of one of their sub-expressions (which may be a non-Boolean value).**
- `X and Y and Z`
  - If all are true, returns value of Z.
  - Otherwise, returns value of first false sub-expression.
- `X or Y or Z`
  - If all are false, returns value of Z.
  - Otherwise, returns value of first true sub-expression.
- ***and* and *or* use *short-circuit evaluation*, so no further expressions are evaluated**

Conditional Expressions

- `x = true_value if condition else false_value`
- Uses short-circuit evaluation:
  - **First, `condition` is evaluated**
  - **If *True*, `true_value` is evaluated and returned**
  - **If *False*, `false_value` is evaluated and returned**
- This looks a lot like C's ternary operator
- Suggested use:

```
x = (true_value if condition else false_value)
```

if, while, assert (i.e., some control flow)

Explicit control-flow constructs

- **There are several Python expressions that control the flow of a program. All of them make use of Boolean conditional tests.**
  - *if* Statements
  - *while* Loops
  - *assert* Statements

if Statements

```
if x == 3:
    print("X equals 3. ")
elif x == 2:
    print("X equals 2. ")
else:
    print("X equals something else. ")
print("This is outside the 'if'. ")
```

Be careful! The keyword *if* is also used in the syntax of filtered *list comprehensions*.

Note:

- **Use of indentation for blocks**
- **Colon (*:*) after boolean expression**

while Loops

```
x = 3
while x < 10:
   x = x + 1
   print("Still in the loop. " )
print("Outside the loop. ")
```

break and *continue*

- You can use the keyword *break* inside a loop to leave the *while* loop entirely.

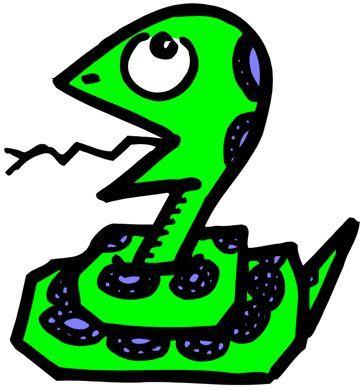
- You can use the keyword *continue* inside a loop to stop processing the current iteration of the loop and to immediately go on to the next one.

assert

- **An *assert* statement will check to make sure that some condition is true during the course of a program.**
  - If the condition if false, the program stops.
  - In addition, the program stops noisily and gives us a line number
  - Sometimes this is called "executable documentation"

```
assert(number_of_players < 5)
```

For Loops



For Loops / List Comprehensions

- **Python′s list comprehensions and split/join operations provide natural idioms that usually require a for-loop in other programming languages.**
  - As a result, Python code uses many fewer for-loops
  - Nevertheless, it′s important to learn about for-loops.

- *Caveat*! **The keywords *for* and *in* are also used in the syntax of list comprehensions, but this is a totally different construction.**

For Loops 1

- A for-loop steps through each of the items in a list, tuple, string, or any other type of object which is "iterable"

```
for <item> in <collection>:
  <statements>
```

- If <collection> is a list or a tuple, then the loop steps through each element of the sequence.
- If <collection> is a string, then the loop steps through each character of the string.

```
for someChar in "Hello World":
   print (someChar)
```

For Loops 2

```
for <item> in <collection>:
    <statements>
```

- **<item> can be more complex than a single variable name.**
  - When the elements of <collection> are themselves sequences, then <item> can match the structure of the elements.
  - This multiple assignment can make it easier to access the individual parts of each element.

```
for (x, y) in [("a",1), ("b",2), ("c",3), ("d",4)]:
    print (x)
```

For loops and the *range()* function

- Since a variable often ranges over some sequence of numbers, the *range()* function returns a list of numbers from 0 up to but not including the number we pass to it.

- range(5) returns [0,1,2,3,4]
- So we could say:

```
for x in range(5):
    print (x)
```

- (There are more complex forms of *range()* that provide richer functionality…)

If you absolutely, positively need an index with your cup o' loops

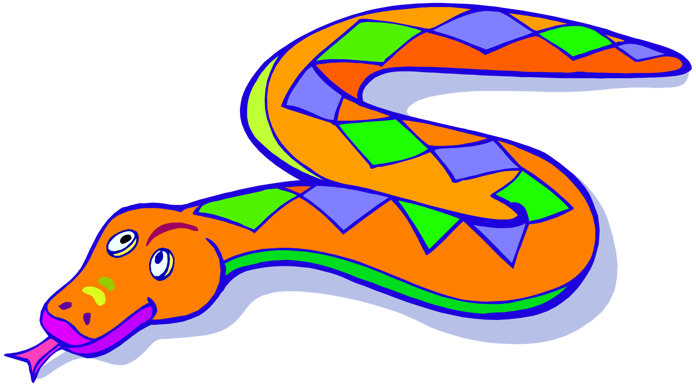
- Use enumerate()
- Returns a sequence of integer, item pairs
- Example

```
for i, item in enumerate(["moo!", "woof!", "meow!", "quack"]):
    print (i, item)

0 moo!
1 woof!
2 meow!
3 quack
```

- This is the best of both worlds (and helps avoid infelicities when the habit is too strong to stop index-based looping)

Generating Lists using ”List Comprehensions”



List Comprehensions

- **A powerful feature of the Python language.**
  - Generate a new list by applying a function to every member of an original list.
  - Python programmers make extensive use of list comprehensions. You′ll see many of them in production code.

- **The syntax of a *list comprehension* is somewhat tricky.**
  - Syntax suggests that of a *for*-loop, an *in* operation, or an *if* statement
    - —all three of these keywords (′*for*′, ′*in*′, and ′*if*′) are also used in the syntax of forms of list comprehensions.

Using List Comprehensions 1

```
>>> li = [3, 6, 2, 7]
>>> [elem*2 for elem in li]
[6, 12, 4, 14]
```

Note: Non-standard colours on next several slides to help clarify the list comprehension syntax.

[expression for name in list]

- Where expression is some calculation or operation acting upon the variable name.

- For each member of the list, the list comprehension
  1. sets name equal to that member,
  2. calculates a new value using expression,
- It then collects these new values into a list which is the return value of the list comprehension.

Using List Comprehensions 2

`[ expression for name in list ]`

- If list contains elements of different types, then expression must operate correctly on the types of all of list members.

- If the elements of list are other containers, then the name can consist of a container of names that match the type and "shape" (or "pattern") of the list members.

```
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [ n * 3 for (x, n) in li]
[3, 6, 21]
```

Using List Comprehensions 3

[**expression** for **name** in **list**]

- **expression** **can also contain user-defined functions.**

```
>>> def subtract(a, b):
        return a - b

>>> oplist = [(6, 3), (1, 7), (5, 5)]
>>> [subtract(y, x) for (x, y) in oplist]
[-3, 6, 0]
```

Filtered List Comprehension 1

```
[expression for name in list if filter]
```

- Filter determines whether expression is performed on each member of the list.
- For each element of list, checks if it satisfies the filter condition.
- If it returns *False* for the filter condition, it is omitted from the list before the list comprehension is evaluated.

Filtered List Comprehension 2

[expression for name in list if filter]

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem * 2 for elem in li if elem > 4]
[12, 14, 18]
```

- **Only 6, 7, and 9 satisfy the filter condition.**
- **So, only 12, 14, and 18 are produced.**

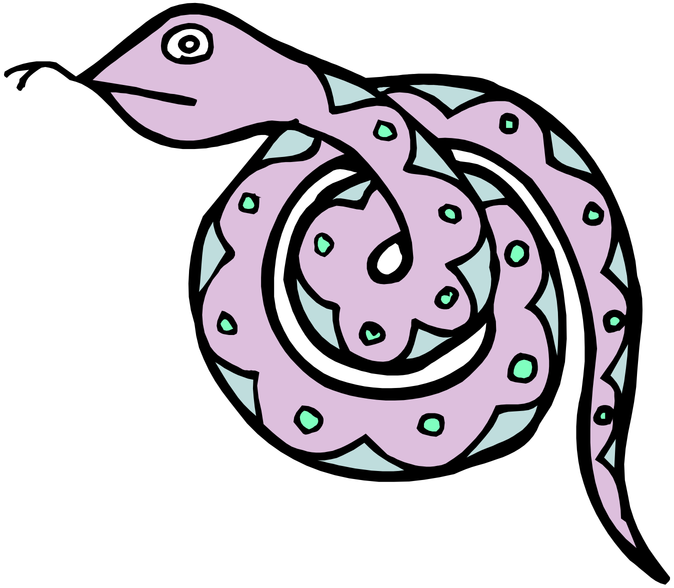
Nested List Comprehensions

- Since list comprehensions take a list as input and produce a list as output, they are easily nested:

```
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in
         [item+1 for item in li] ]
[8, 6, 10, 4]
```

- The inner comprehension produces: [4, 3, 5, 2].
- So, the outer one produces: [8, 6, 10, 4].

Some Fancy Function Syntax



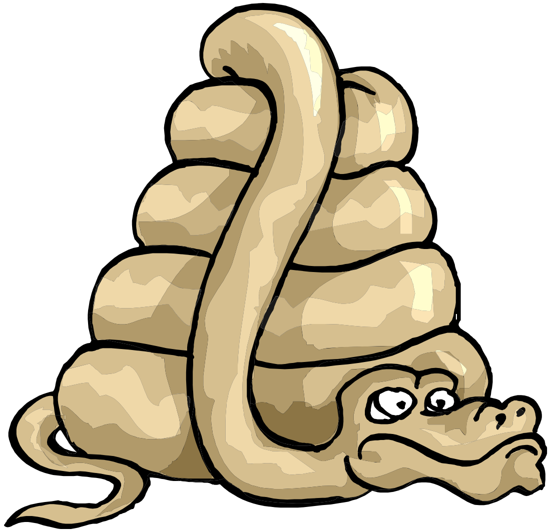
Lambda Notation

- **Functions can be defined without giving them names.**
- **This is most useful when passing a short function as an argument to another function.**

```
>>> def applier(q, x):
       return q(x)
>>> applier(lambda z: z * 4, 7)
  28
```

- **The first argument given to applier() is an unnamed function that takes one input and returns the input multiplied by four.**
- **Note: only single-parameter, single-expression functions can be defined using this lambda notation.**
- **Lambda notation has a rich history in program language research, AI, and the design of the LISP language.**

String Conversions & String Operations



String to List to String

- join turns a list of strings into one string.

  <separator_string>.join(<some_list>)

```
>>> ":".join( ["abc", "def", "ghi"] )
 "abc:def:ghi"
```

- split turns one string into a list of strings.

  <some_string>.split(<separator_string>)

```
>>> "abc:def:ghi".split( ":" )
 ["abc", "def", "ghi"]
```

- Note the inversion in the syntax

Convert Anything to a String

- **The built-in str() function can convert an instance of any data type into a string.**

  You can define how this function behaves for user-created data types. You can also redefine the behavior of this function for many types.

```
>>>  "Hello " + str(2)
"Hello 2"
```

String Operations

- A number of methods for the string class perform useful formatting operations:

```
>>> "hello".upper()
'HELLO'
```

- Check the Python documentation for many other handy string operations.

- Helpful hint: use `<string>.strip()` to strip off final newlines from lines read from files

String Formatting Operator: %

- **The operator % allows strings to be built out of many data items in a "fill in the blanks" fashion.**
  - Allows control of how the final string output will appear.
  - For example, we could force a number to display with a specific number of digits after the decimal point.

- **Very similar to the sprintf command of C.**

```
>>> x = "abc"
>>> y = 34
>>> "%s xyz %d" % (x, y)
'abc xyz 34'
```

- **The tuple following the % operator is used to fill in the blanks in the original string marked with *%s* or *%d*.**
  - **Check Python documentation for whether to use %s, %d, or some other formatting code inside the string.**

Printing with Python

- You can print a string to the screen using "print".
- Using the % string operator in combination with the print command, we can format our output text.

```
>>> print("%s xyz %d"  %  ("abc", 34) )
```

```
abc xyz 34
```

"print" automatically adds a newline to the end of the string. If you include a list of strings, it will concatenate them with a space between them.

```
>>> print("abc")
```

```
abc
```

```
>>> print("abc", "def")
```

```
abc def
```

- Useful: `>>> print("abc", end = " ")` doesn't add newline just a single space.

More complex formatting

- Sometimes we want tight control over the way our string are output
- Strings are objects and therefore respond to messages, including format()
- Idea: string template (w/ format & positions) + arguments

```
>>> print ('Course unit: {}; Number {}'.format('SENG', '265'))
Course unit: SENG; Number 265

>>> print ('Course unit: {0}; Number {1}'.format('SENG', '265'))
Course unit: SENG; Number 265

>>> print ('Course unit: {1}; Number {0}'.format('265', 'SENG'))
Course unit: SENG; Number 265

>>> print ('Course unit: {1}; & again {1}'.format('265', 'SENG'))
Course unit: SENG; & again SENG
```

More complex formatting

- **Can control the size of numeric fields**

```
>>> import math
>>> print ('Value of e is about {0:.3f}'.format(math.e))
Value of e is about 2.718

>>> print ('{0:0>4} {1:0<4} {2:0^4}'.format(11, 22, 33))
0011 2200 0330
```

- **For more string-formatting wizardry visit:**

  https://docs.python.org/3/library/string.html

"mywc.py": one approach

```
#!/usr/bin/env python

import sys

def main():
    num_chars = 0
    num_words = 0
    num_lines = 0

    for line in sys.stdin:
        num_lines = num_lines + 1
        num_chars = num_chars + len(line)
        line =  line.strip()
        words = line.split()
        num_words = num_words + len(words)

    print (num_lines, num_words, num_chars)


if __name__ == "__main__":
    main()
```

"mywc.py": stdin or filename?

```
#!/usr/bin/env python

import fileinput
import sys

def main():
    num_chars = 0
    num_words = 0
    num_lines = 0

    for line in fileinput.input():
        num_lines = num_lines + 1
        num_chars += len(line)
        line = line.strip()
        words = line.split()
        num_words += len(words)

    print (num_lines, num_words, num_chars)


if __name__ == "__main__":
    main()
```

If filenames are provided to the script, this loop will iterate through all lines in all of the files.

If no filename is provided, the loop will iterate through all lines in stdin.

"mywc.py": a contrived "while" loop

```
#!/usr/bin/env python

import sys

def main():
    num_chars = 0
    num_words = 0
    num_lines = 0

    lines = sys.stdin.readlines()

    while (lines):
        a_line = lines[0]
        num_lines = num_lines + 1
        num_chars += len(a_line)
        a_line = a_line.strip()
        words = a_line.split()
        num_words += len(words)
        lines = lines[1:]

    print (num_lines, num_words, num_chars)


if __name__ == "__main__":
    main()
```

This line using "readlines() could lead to indigestion if the input is very large...

Note the difference between accessing the head of a list...

and accessing the tail of a list...