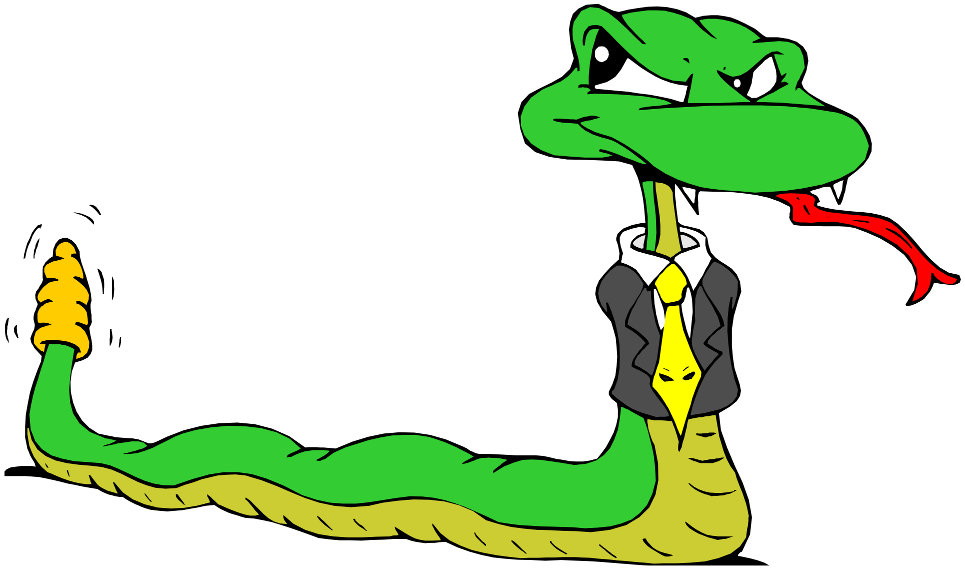
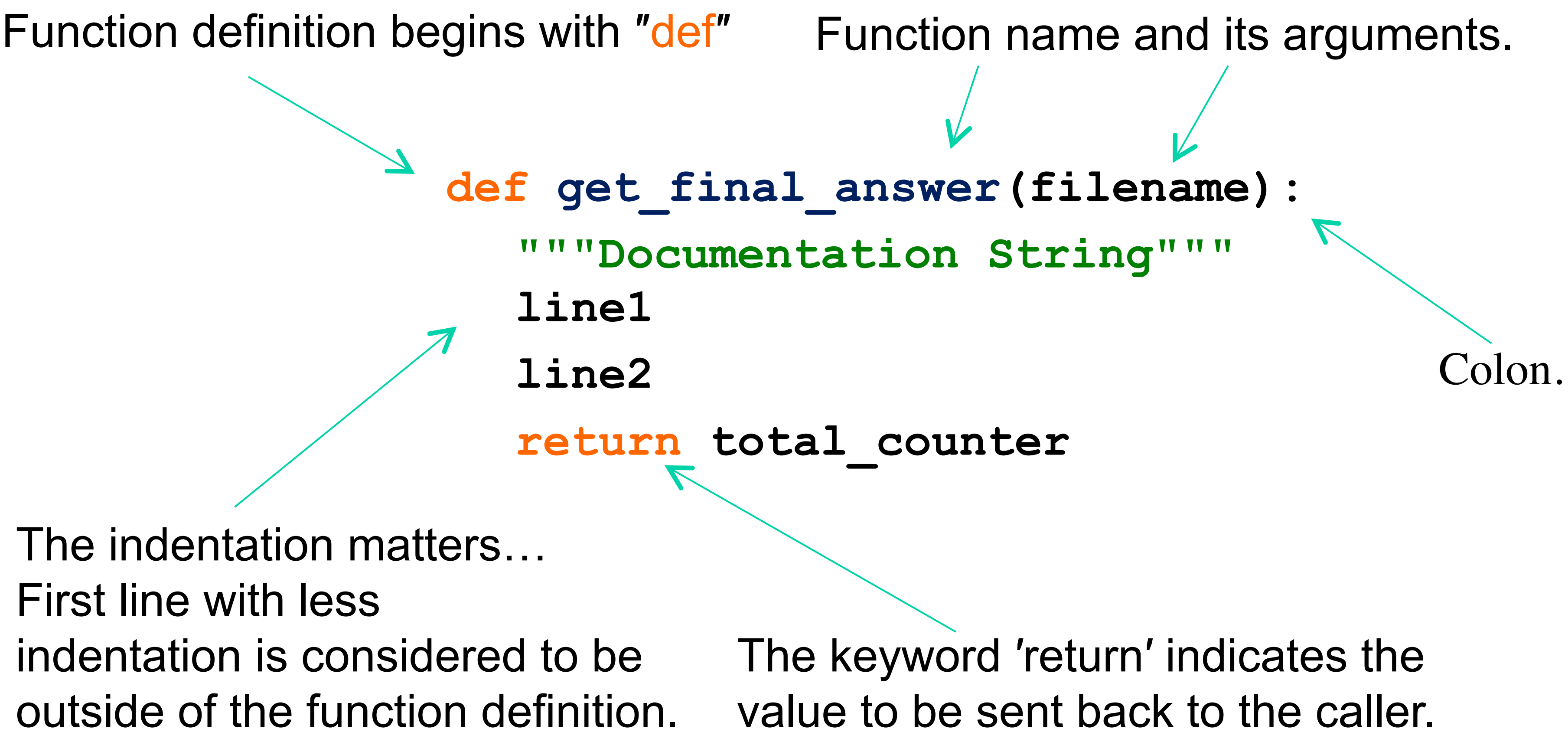
Functions in Python



Defining Functions



Function definition begins with "def"

Function name and its arguments.

```
def get_final_answer(filename):
    """Documentation String"""
    line1
    line2
    return total_counter
```

Colon.

The indentation matters…
First line with less
indentation is considered to be
outside of the function definition.

The keyword 'return' indicates the
value to be sent back to the caller.

No header file or declaration of types of function or arguments.

Python and Types

Python determines the data types of *variable bindings* in a program automatically.

"Dynamic Typing"

But Python's not casual about types, it enforces the types of *objects*.

"Strong Typing"

So, for example, you can't just append an integer to a string. You must first convert the integer to a string itself.

```
x = "the answer is "   # Deduces x is bound to a string.
y = 23                 # Deduces y is bound to an integer.
print x + y            # Python will complain about this.
```

Calling a Function

- **The syntax for a function call is:**

```
>>> def myfun(x, y):
            return x * y
>>> myfun(3, 4)
12
```

- **Parameters in Python are “Call by Assignment.”**
  - Sometimes acts like “call by reference” and sometimes like “call by value” in C++.
    - —Mutable datatypes: Behaves like Call-by-reference.
    - —Immutable datatypes: Behaves like Call-by-value.

Functions without returns

- ***All* functions in Python have a return value**
  - even if no *return* line inside the code.
- **Functions without a *return* actually do return the special value *None*.**
  - *None* is a special constant in the language.
  - *None* is used like *NULL*, *void*, or *nil* in other languages.
  - *None* is also logically equivalent to False.
  - The interpreter doesn't print *None*

Function overloading? No.

- **There is no function overloading in Python.**
  - Unlike C++, a Python function is specified by its name alone
    - — The number, order, names, or types of its arguments cannot be used to distinguish between two functions with the same name.
  - Two different functions can't have the same name, even if they have different arguments.
- **But: see *operator overloading* in later slides**

Functions are first-class objects in Python

- Functions can be used as any other data type
- They can be
  - Arguments to function
  - Return values of functions
  - Assigned to variables
  - Parts of tuples, lists, etc
  - …

```
>>> def myfun(x):
        return x*3

>>> def applier(q, x):
        return q(x)

>>> applier(myfun, 7)
21
```

Slight detour: "main" function

```
#!/usr/bin/python

def main():
    print "Here we are in main. About to visit caveOfCaerbannog."
    caveOfCaerbannog()
    print
    print "Now we're back in main. About to call camelot()."
    camelot()
    print
    print "I feel happy! I feel hap..."

def caveOfCaerbannog():
    print "We are visiting the dreadful Cave of Caerbannog."
    print "Heck, there are cute rabbits here like at UVic."
    print "Come here little raaaaa... AUGH!"

def camelot():
    print "Here we are in Camelot."
    print "Let's leave. It's too silly here."

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

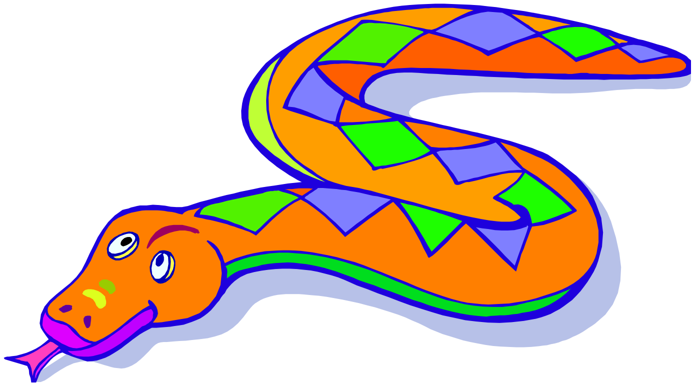
Another detour: command-line args

```
#!/usr/bin/python

import optparse

def main():
    p = optparse.OptionParser()
    p.add_option('--width', '-w', default=75)
    p.add_option('--indent', '-i', default=0)
    p.add_option('--number', '-n', action='store_true')

    options, arguments = p.parse_args()

    print "Width: ", options.width
    print "Indent: ", options.indent
    print "Numbering: ", options.number

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

Another detour: command-line args

```
#!/usr/bin/python

import optparse

def main():
    p = optparse.OptionParser()

    # You need not specify options!
    #
    options, arguments = p.parse_args()
    if len(arguments) == 0:
        print "No arguments"
    else:
        print "Argument is ", arguments[0]

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

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)
```

Control Flow

Control of Flow

- **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.

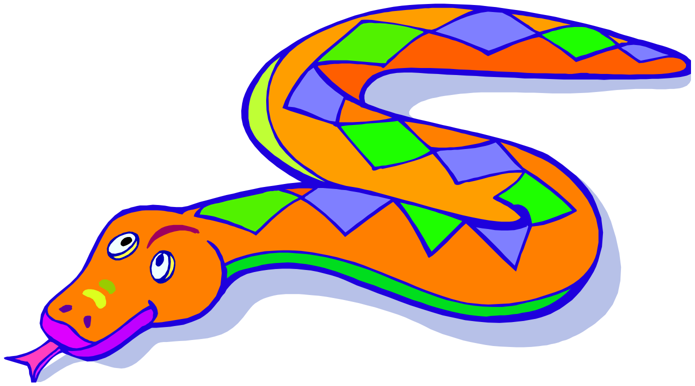
- You can use the keyword *continue* inside a loop to stop processing the current iteration of the loop and to immediately go on to the next one.

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)
```

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 use list comprehensions extensively. You'll see many of them in live 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 colors 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.

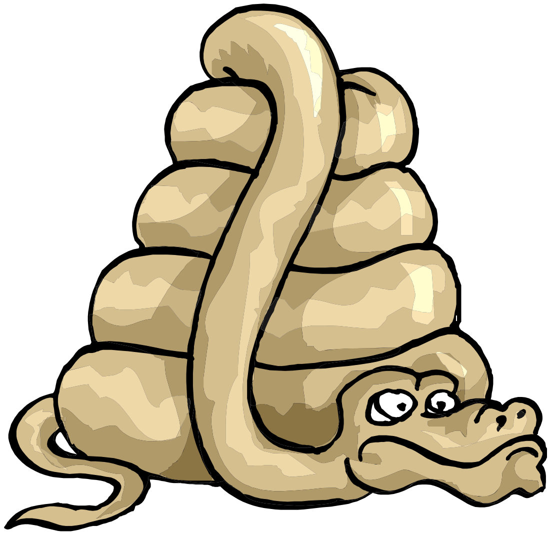
Nested List Comprehensions

- 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].

String Conversions



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