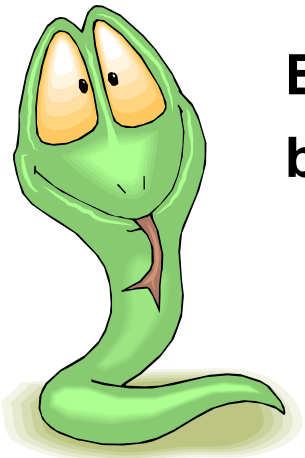

Python: A Simple Tutorial

**Adapted from slides for CIS 391 by Matt Huenerfauth
University of Pennsylvania**



**Edited further for UVic SENG 265
by Nigel Horspool & Michael Zastre (University of Victoria)**

Python

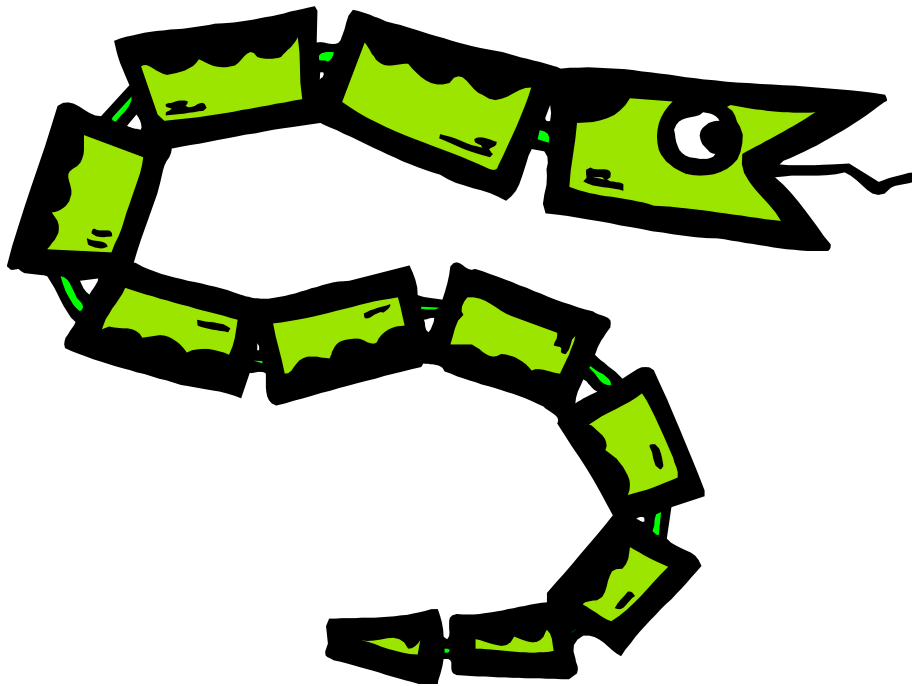
- Python is an open source scripting language.
- Developed by Guido van Rossum in the early 1990s
- Named after Monty Python
- Available for download from <http://www.python.org>
- (We're using Python 3.6.8 in this course, although the most stable recent release is 3.10.7)



Why Python?

- **Supports Object-Oriented style of programming...**
- **... but you don't always need to use it**
- **Much less verbose than Java**
- **Cleaner syntax than Perl**
- **Built-in datatypes for strings, lists, and more**
- **Strong numeric processing capabilities: matrix operations, etc. (and more via numpy, pandas, scikit, pandas)**
- **Suitable for experimenting with machine-learning code**
- **Powerful implementation of a regular-expressions library**

The Basics



A Code Sample

```
x = 34 - 23          # A comment
y = "Hello"          # Another one.
z = 3.45
w = 0.9

if z == 3.45 or y == "Hello":
    x = x + 1          # Addition
    y = y + " World!"  # String concatenation
    w = x // 4          # Integer division
    z = x / 4           # Floating point division

print (x)
print (format(y, ".2f")) # Two digits after decimal
```

Understanding the Code...

- Assignment uses `=` and comparison uses `==`.
- For numbers `+` `-` `*` `/` `%` behave as expected.
 - Special use of `+` for string concatenation.
 - Special use of `%` for string formatting (as with `printf` in C)
- Logical operators are words (`and`, `or`, `not`) *which is different that* used in C or Java (i.e., do not use `&&`, `||`, `!`)
- The basic printing function is `print`.
- The first assignment to a variable creates it.
 - Variable types don't need to be declared.
 - Python figures out the variable types on its own.
- Block structure is denoted by indentation.

Basic Datatypes

- **Integers (default for numbers)**

`z = 5 // 2` `# Answer is 2, integer division.`

- **Floats**

`x = 3.456`

`y = 4 / 3` `# Answer is 1.33...`
 `# floating-point division`

- **Strings**

- Can use double- or single-quotes to delimit strings.

`"abc"` `'abc'` (Same thing.)

- Unmatched quotation marks can occur within the string.

`"matt's"`

- Use triple double-quotes for multi-line strings or strings than contain both ' and " inside them:

`"""a'b'c"""`

Whitespace

White space is meaningful in Python: especially indentation and placement of newlines.

- Use a newline to end a line of code.
 - Use `\` when must go to next line prematurely.
- No braces `{ }` to mark blocks of code in Python...
Use *consistent* indentation instead.
 - The first line with *less* indentation is outside the block.
 - The first line with *more* indentation starts a nested block
- Often a colon appears at the start of a new block.
(E.g. for function and class definitions.)
- Tip: Configure your editor to use spaces for indents (i.e., not tabs!)

Comments

- Start comments with # – the rest of line is ignored.
- (This is a bit like `"/"` in Java and C++)
- Can include a "documentation string" as the first line of any new function or class that you define.
- The development environment, debugger, and other tools make use of such documentation strings, therefore it is good style to include one.

```
def my_function(x, y):  
    """This is the docstring. This here function  
    does something truly wonderful, or so  
    we think despite seeing no code at all."""  
    # The code would go here...
```

Assignment

- **Binding a variable in Python means setting a *name* to hold a *reference* to some *object*.**
 - *Assignment creates references, not copies*
- **Names in Python do not have an intrinsic type. Objects have types.**
 - Python determines the type of the reference automatically based on the data object assigned to it.
- **You create a name the first time it appears on the left side of an assignment expression:**
`x = 3`
- **A reference is deleted via garbage collection after any names bound to it have passed out of scope.**

Accessing Non-Existent Names

If you try to access a name before it's been properly created (by placing it on the left side of an assignment), you'll get an error.

```
>>> y
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#16>", line 1, in -toplevel-
```

```
    y
```

```
NameError: name 'y' is not defined
```

```
>>> y = 3
```

```
>>> y
```

```
3
```

Multiple Assignment

You can also assign to multiple names at the same time.

```
>>> x, y = 2, 3
```

```
>>> x
```

```
2
```

```
>>> y
```

```
3
```

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

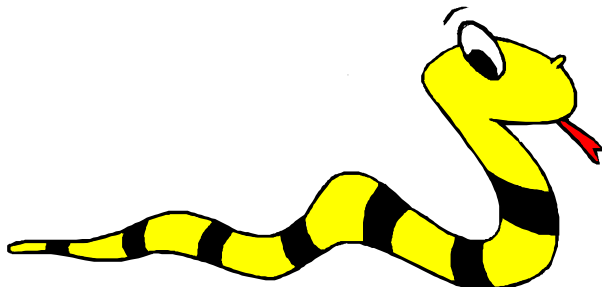
bob Bob _bob _2_bob_ bob_2 BoB

- There are some reserved words:

and, assert, break, class, continue, def, del,
elif, else, except, exec, finally, for, from,
global, if, import, in, is, lambda, not, or, pass,
print, raise, return, try, while

Sequence types:

Tuples, Strings, and Lists



Sequence Types

1. Tuple

- A simple *immutable* ordered sequence of items
- Items can be of mixed types, including collection types

2. Strings

- *Immutable*
- **Conceptually very much like a tuple**

3. List

- *Mutable* ordered sequence of items of mixed types

Similar Syntax

- All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.
- Key difference:
 - Tuples and strings are *immutable* (cannot be modified / changed in place)
 - Lists are *mutable* (can be modified / changed in place)
- The operations shown in this section can be applied to *all* sequence types
 - most examples will just show the operation performed on one

Sequence Types 1

- Tuples are defined using parentheses (and commas).

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Lists are defined using square brackets (and commas).

```
>>> li = ["abc", 34, 4.34, 23]
```

- Strings are defined using quotes – *single*, *double*, or *triple* ('', "", or """).

```
>>> st = "Hello World"
```

```
>>> st = 'Hello World'
```

```
>>> st = """This is a multi-line  
string that uses triple quotes."""
```

Sequence Types 2

- We can access individual members of a tuple, list, or string using square bracket "array" notation.
- *Note that all are 0 based...*

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]      # Second item in the tuple.
'abc'
```

```
>>> li = ["abc", 34, 4.34, 23]
>>> li[1]      # Second item in the list.
34
```

```
>>> st = "Hello World"
>>> st[1]      # Second character in string.
'e'
```

Positive and negative indices

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0.

```
>>> t[1]  
'abc'
```

Negative lookup: count from right, starting with -1.

```
>>> t[-3]  
4.56
```

Slicing: Return Copy of a Tuple (part 1)

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying before the second index.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

You can also use negative indices when slicing.

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Slicing: Return Copy of a Tuple (part 2)

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Omit the first index to make a copy starting from the beginning of the container.

```
>>> t[:2]  
(23, 'abc')
```

Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> t[2:]  
(4.56, (2,3), 'def')
```

Copying the Whole Sequence

To make a *copy* of an entire sequence, you can use `[:]`.

```
>>> li[:]  
[23, 'abc', 4.56, (2,3), 'def']
```

Note the difference between these two lines for mutable sequences:

```
>>> list2 = list1          # 2 names refer to 1 ref  
                             # Changing one affects both  
  
>>> list2 = list1[:]       # Two independent copies,  
                             # two refs
```

The 'in' Operator

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- For strings, tests for substrings

```
>>> s = 'abcde'
>>> 'c' in s
True
>>> 'cd' in s
True
>>> 'ac' in s
False
```

- Be careful: the *in* keyword is also used in the syntax of *for loops* and *comprehensions*.

The + Operator

- The + operator produces a *new* tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
```

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

```
>>> "Hello" + " " + "World"
'Hello World'
```

```
>>> "Hello", "World"
('Hello', 'World')
```


The * Operator

- The * operator produces a *new* tuple, list, or string that "repeats" the original content.

```
>>> (1, 2, 3) * 3  
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

```
>>> [1, 2, 3] * 3  
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
>>> "Hello" * 3  
'HelloHelloHello'
```

Mutability: **Tuples vs. Lists**



Tuples: Immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14
```

```
Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

You cannot change a tuple.

However, you can make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

Lists: Mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
['abc', 45, 4.34, 23]
```

- We can change lists *in place*.
- Name *li* still points to the same memory reference when we are done.
- The mutability of lists means that operations on lists are not as fast as operations on tuples.

Operations on Lists Only

```
>>> li = [1, 11, 3, 4, 5]
```

```
>>> li.append('a')    # Our first exposure to method syntax
```

```
>>> li
```

```
[1, 11, 3, 4, 5, 'a']
```

```
>>> li.insert(2, 'i')
```

```
>>> li
```

```
[1, 11, 'i', 3, 4, 5, 'a']
```

The *extend* method vs the **+** operator.

- **+** creates a fresh list (with a new memory reference)
- *extend* operates on list **li** in place.

```
>>> li.extend([9, 8, 7])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

Confusing:

- Extend takes a list as an argument.
- Append takes a singleton as an argument.

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

extend != append

Operations on Lists Only

```
>>> li = ['a', 'b', 'c', 'b']
```

```
>>> li.index('b')      # index of first occurrence  
1
```

```
>>> li.count('b')      # number of occurrences  
2
```

```
>>> li.remove('b')     # remove first occurrence  
>>> li  
['a', 'c', 'b']
```

Operations on Lists Only

```
>>> li = [5, 2, 6, 8]
```

```
>>> li.reverse()      # reverse the list *in place*
```

```
>>> li  
[8, 6, 2, 5]
```

```
>>> li.sort()         # sort the list *in place*
```

```
>>> li  
[2, 5, 6, 8]
```

```
>>> li.sort(some_function)  
# sort in place using user-defined comparison
```

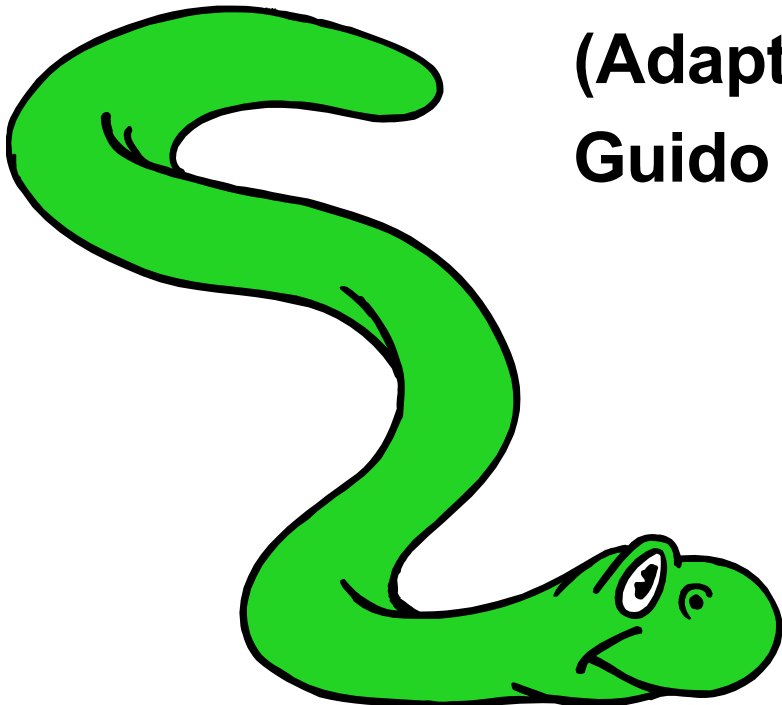

Tuples vs. Lists

- **Lists are slower at runtime, but more flexible than tuples.**
 - Lists can be modified, and they have lots of handy operations we can perform on them.
 - Tuples are immutable and have fewer features.
- **To convert between tuples and lists use the `list()` and `tuple()` functions:**

```
li = list(tu)
tu = tuple(li)
```

Understanding Reference Semantics in Python

(Adapted from several slides by
Guido van Rossum)



Understanding Reference Semantics

- **Assignment manipulates references**

`x = y` **does not make a copy** of the object `y` references

`x = y` makes `x` **reference** the object `y` references

- **Very useful; but beware!**

- **Example:**

```
>>> a = [1, 2, 3] # a now references the list [1, 2, 3]
```

```
>>> b = a        # b now references what a references
```

```
>>> a.append(4)   # this changes the list a references
```

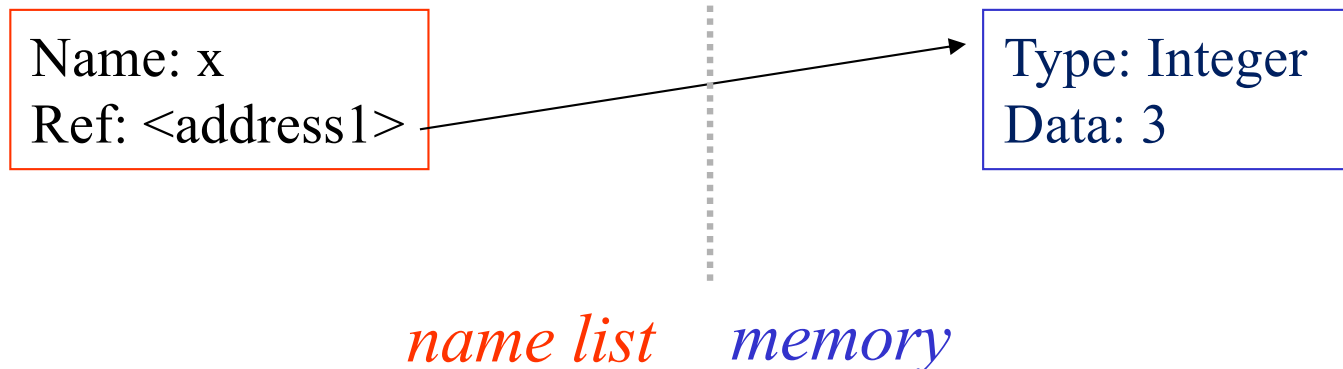
```
>>> print(b)      # if we print what b references,
```

```
[1, 2, 3, 4]      # SURPRISE! It has changed...
```

Why??

Understanding Reference Semantics II

- There is a lot going on when we type:
`x = 3`
- First, an integer **3** is created and stored in memory
- A name **x** is created
- A **reference** to the memory location storing the **3** is then assigned to the name **x**
- So: When we say that the value of **x** is **3**
- we mean that **x** now refers to the integer **3**



Understanding Reference Semantics III

- The data 3 we created is of type integer. In Python, the datatypes integer, float, and string (and tuple) are "immutable."
- This does not mean we cannot change the value of x, i.e. *change what x refers to ...*
- For example, we could increment x:

```
>>> x = 3
```

```
>>> x = x + 1
```

```
>>> print (x)
```

```
4
```

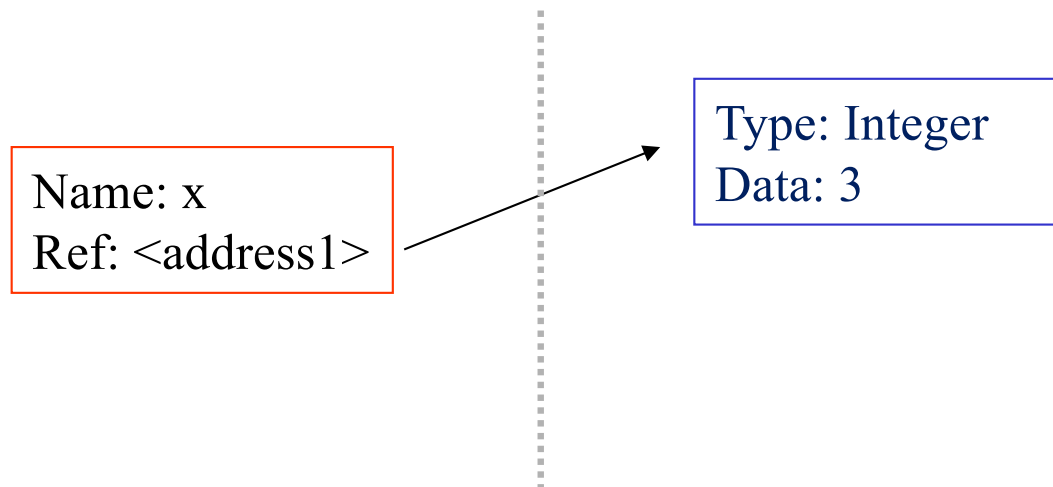
Understanding Reference Semantics IV

- If we increment x , then what's really happening is:

1. *The reference of name x is looked up.*

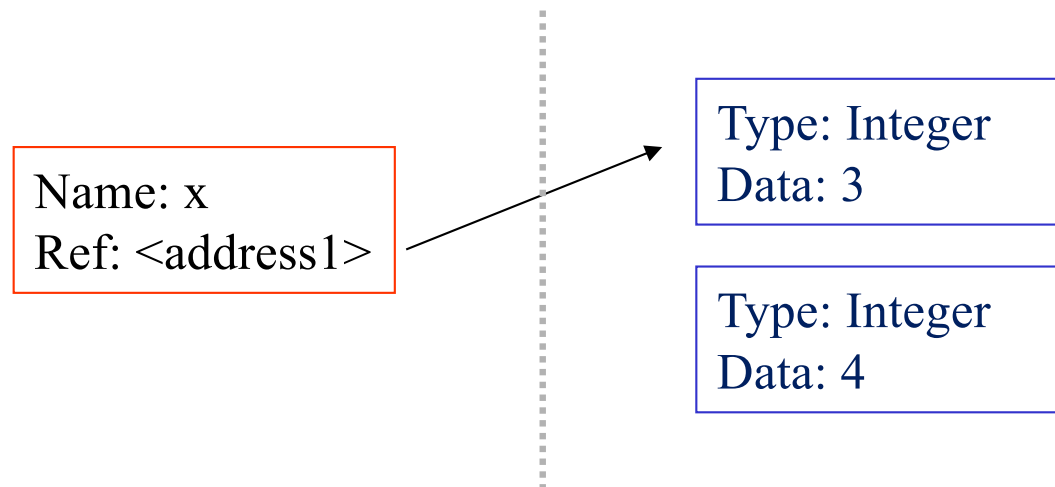
`>>> x = x + 1`

2. *The value at that reference is retrieved.*



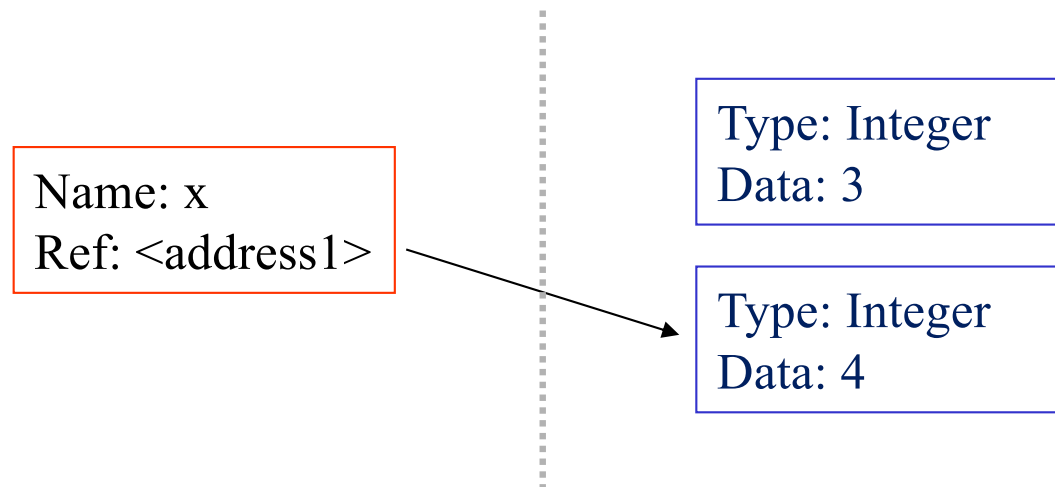
Understanding Reference Semantics IV

- If we increment x , then what's really happening is:
 1. The reference of name x is looked up. `>>> x = x + 1`
 2. The value at that reference is retrieved.
 3. *The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.*



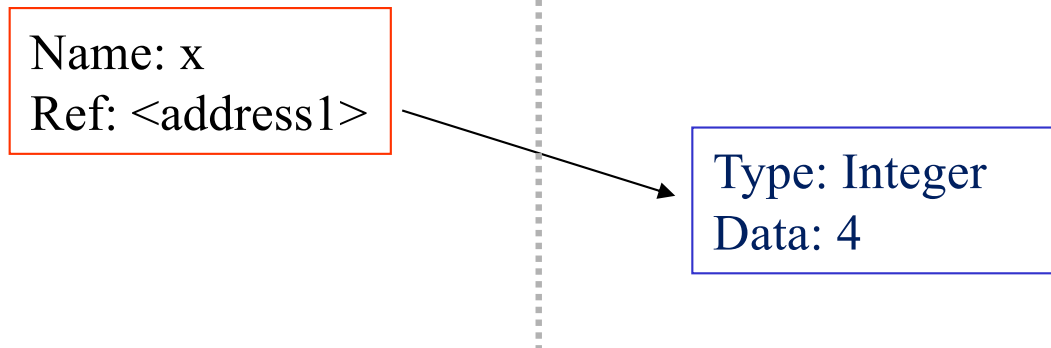
Understanding Reference Semantics IV

- If we increment x , then what's really happening is:
 1. The reference of name x is looked up. `>>> x = x + 1`
 2. The value at that reference is retrieved.
 3. The $3+1$ calculation occurs, producing a new data element **4** which is assigned to a fresh memory location with a new reference.
 4. *The name x is changed to point to this new reference.*



Understanding Reference Semantics IV

- If we increment x , then what's really happening is:
 1. The reference of name x is looked up. `>>> x = x + 1`
 2. The value at that reference is retrieved.
 3. The $3+1$ calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
 4. The name x is changed to point to this new reference.
 5. *The old data 3 is garbage collected if no name still refers to it.*



Assignment (part 1)

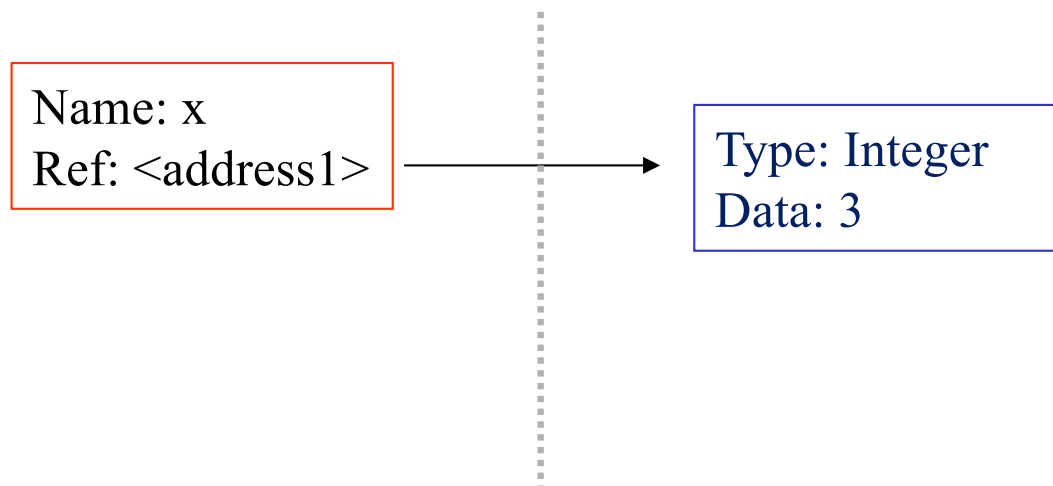
- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print(x)       # No effect on x, still ref 3.
3
```

Assignment (part 1)

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

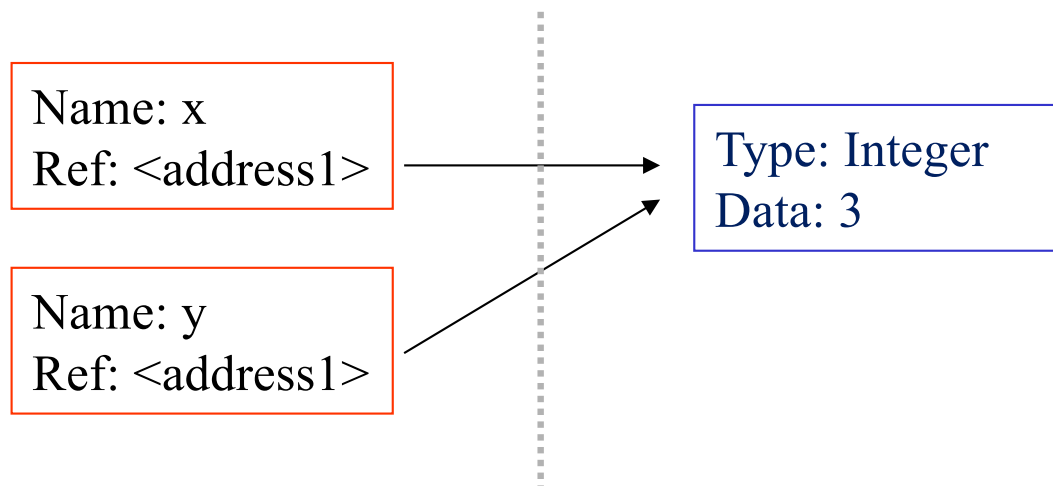
```
→ >>> x = 3          # Creates 3, name x refers to 3
   >>> y = x          # Creates name y, refers to 3.
   >>> y = 4          # Creates ref for 4. Changes y.
   >>> print(x)       # No effect on x, still ref 3.
3
```



Assignment (part 1)

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

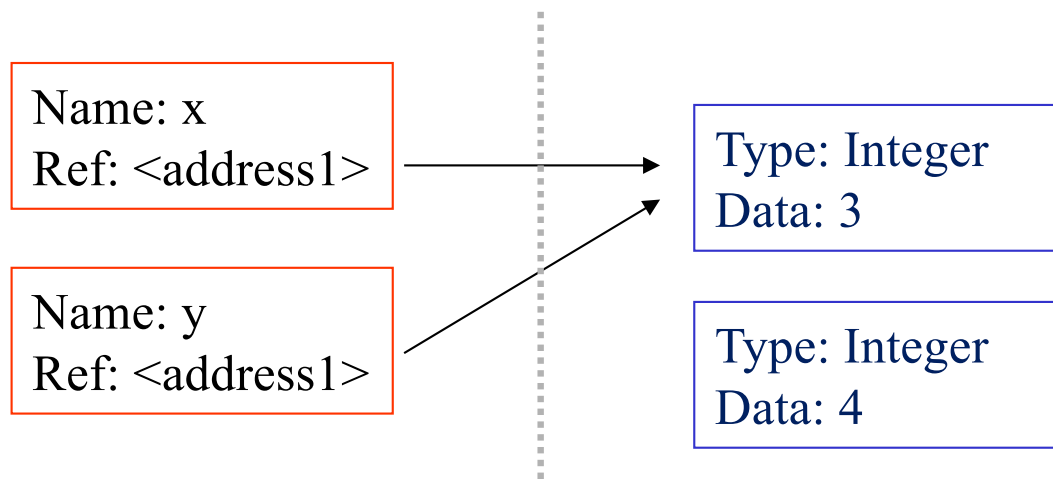
```
>>> x = 3          # Creates 3, name x refers to 3
→ >>> y = x         # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print(x)       # No effect on x, still ref 3.
3
```



Assignment (part 1)

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

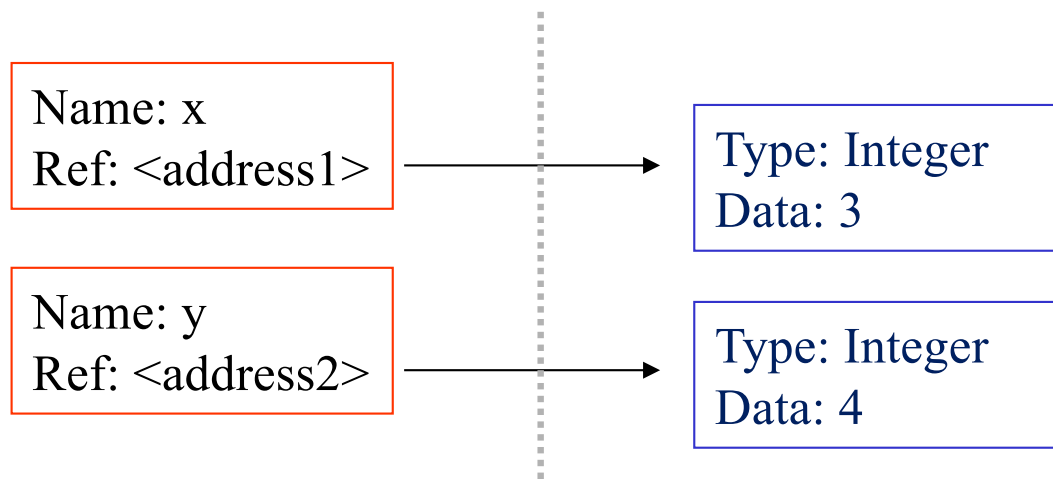
```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
→>>> y = 4         # Creates ref for 4. Changes y.
>>> print(x)       # No effect on x, still ref 3.
3
```



Assignment (part 1)

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

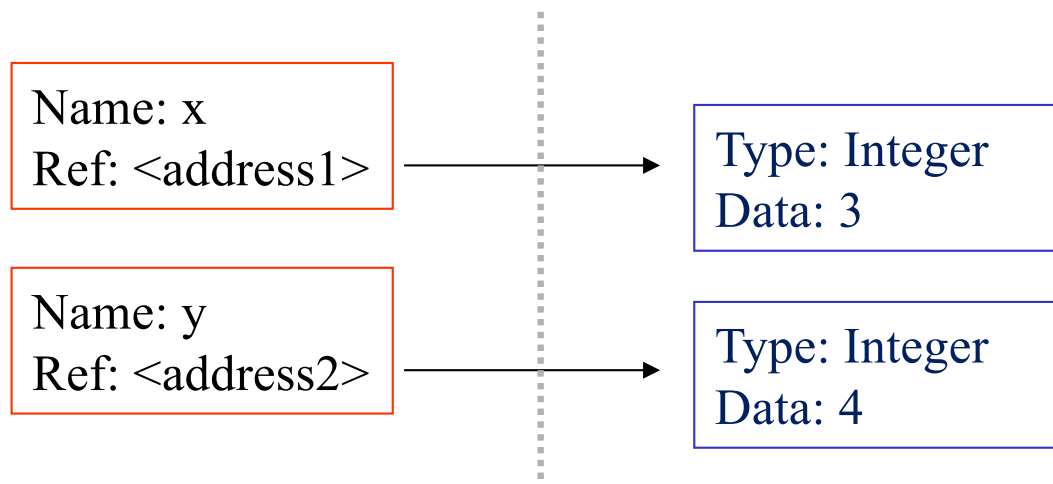
```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
→>>> y = 4         # Creates ref for 4. Changes y.
>>> print(x)       # No effect on x, still ref 3.
3
```



Assignment (part 1)

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
→>>> print(x)      # No effect on x, still ref 3.
3
```



Assignment (part 2)

- **For some other data types (lists, dictionaries, user-defined types), assignment works differently.**
 - These datatypes are **"mutable."**
 - When we change these data, we do it *in place*.
 - We don't copy them into a new memory address each time.
 - If we type `y=x` and then modify `y`, both `x` and `y` are changed.

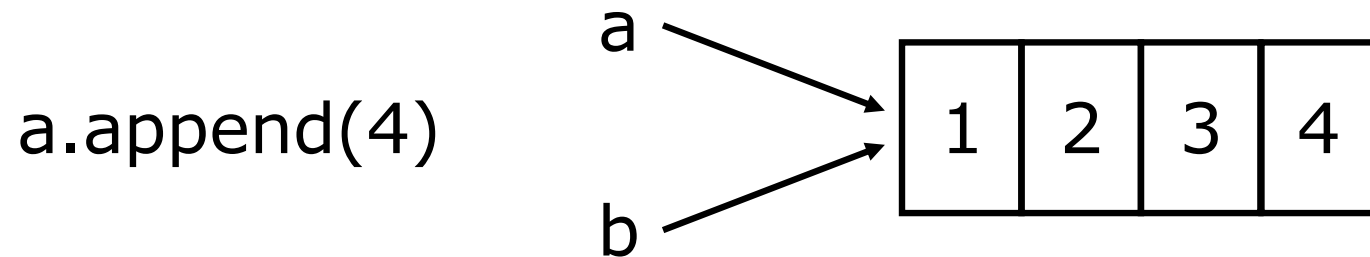
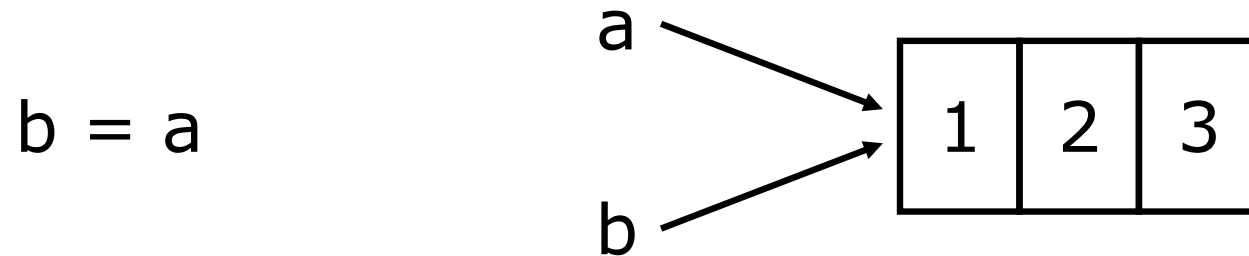
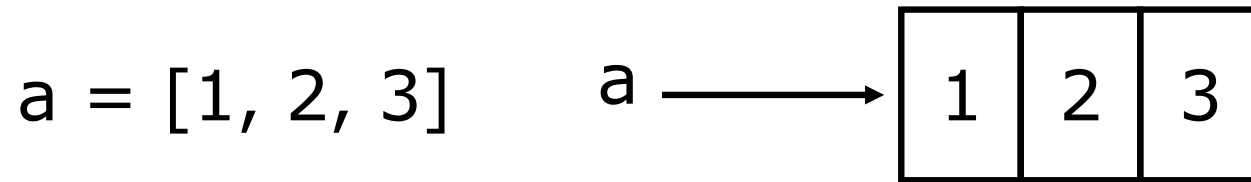
immutable

```
>>> x = 3
>>> y = x
>>> y = 4
>>> print x
3
```

mutable

```
x = some mutable object
y = x
make a change to y
look at x
x will be changed as well
```


Why? Changing a Shared List

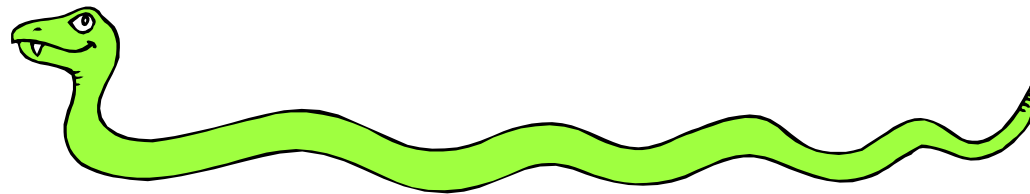


Our surprising example surprising no more...

- So now, here's our code:

```
>>> a = [1, 2, 3] # a now references the list [1, 2, 3]
>>> b = a        # b now references what a references
>>> a.append(4)   # this changes the list a references
>>> print (b)     # if we print what b references,
[1, 2, 3, 4]      # SURPRISE! It has changed...
```

Dictionaries



Dictionaries: A *Mapping* type

- Dictionaries store a *mapping* between a set of keys and a set of values.
 - Keys can be any *immutable* type.
 - Values can be any type
 - A single dictionary can store values of different types
- You can define, modify, view, lookup, and delete the key-value pairs in the dictionary.

Creating and accessing dictionaries

```
>>> d = {'user': 'bozo', 'pswd': 1234}
```

```
>>> d['user']  
'bozo'
```

```
>>> d['pswd']  
1234
```

```
>>> d['bozo']
```

```
Traceback (innermost last):  
  File '<interactive input>' line 1, in ?  
KeyError: bozo
```

Updating Dictionaries

```
>>> d = {'user': 'bozo', 'pswd': 1234}

>>> d['user'] = 'clown'
>>> d
{'user': 'clown', 'pswd': 1234}
```

- Keys must be unique.
- Assigning to an existing key replaces its value.

```
>>> d['id'] = 45
>>> d
{'user': 'clown', 'id': 45, 'pswd': 1234}
```

- Dictionaries may be unordered depending on the version of Python used
 - Unless you are absolutely sure of the version of Python will be executing your code, it is perhaps best to avoid depending upon key-storage order...
- (Dictionaries work by *hashing*)

Removing dictionary entries

```
>>> d = {'user': 'bozo', 'p': 1234, 'i': 34}

>>> del d['user']           # Remove one key/value pair.
>>> d
{'p': 1234, 'i': 34}

>>> d.clear()              # Remove all key/value pairs.
>>> d
{}
```

Useful Accessor Methods

```
>>> d = {'user': 'bozo', 'p': 1234, 'i': 34}

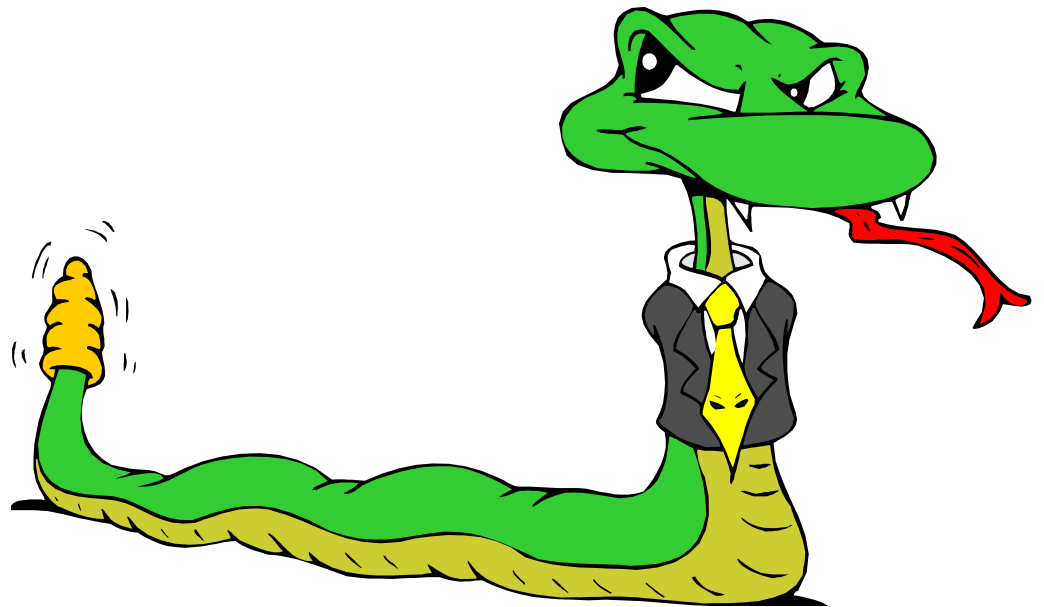
>>> d.keys()                # "List" of keys.
dict_keys(['user', 'p', 'i']) # Order: YMMV

>>> list(d.keys())          # Actual list
['user', 'p', 'i']

>>> d.values()              # List of values.
dict_values(['bozo', 1234, 34])

>>> d.items()               # List of item tuples.
dict_items([('user', 'bozo'), ('p', 1234), ('i', 34)])
```

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.

Reminder: 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* later in the course's treatment of Python**

Functions are first-class objects in Python

- Functions can be used as any other data type
- If something is a first-object, it can be:
 - an argument to a function
 - a return values from a function
 - assigned to a variable
 - a part of a tuple, list, or any other containers

```
>>> def myfun(x):  
        return x*3  
  
>>> def applier(q, x):  
        return q(x)  
  
>>> applier(myfun, 7)  
21
```

Default Values for Arguments

- You can provide default values for a function's arguments
- These arguments are optional when the function is called

```
>>> def myfun(b, c=3, d="hello") :  
        return b + c  
>>> myfun(5, 3, "hello")  
>>> myfun(5, 3)  
>>> myfun(5)
```

Each function call above returns 8.

The Order of Arguments

- You can call a function with some or all of its arguments out of order as long as you specify them (these are called keyword arguments). You can also just use keywords for a final subset of the arguments.

```
>>> def myfun(a, b, c):  
        return a-b  
>>> myfun(2, 1, 43)  
1  
>>> myfun(c=43, b=1, a=2)  
1  
>>> myfun(2, c=43, b=1)  
1
```

A detour: command-line args

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

```
import sys
```

```
def main():
```

```
    if len(sys.argv) == 1:  
        print ("No arguments")
```

```
    else:
```

```
        print ("First argument is", sys.argv[1])
```

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

We will eventually look at the meaning and forms of the **import** statement. For now, however, you consider it as having roughly the same meaning as in Java.

This approach to retrieving command-line arguments is a bit distasteful given we can use the `argparse` module.

However, sometimes you do want the raw arguments/options, and this approach will give them to you in the `sys.argv` array.

A detour: command-line args

```
#!/usr/bin/env python # Don't need to guess bang path for "python"

import argparse

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--width', type=int, default=75,
                        help='length of line')
    parser.add_argument('--indent', type=int, default=0,
                        help='number of blank characters to use at start of line')
    parser.add_argument('--number', type=int, default=10,
                        help="add line numbering")
    parser.add_argument('filename', nargs='?', help='file to be processed')

    args = parser.parse_args()

    print ('width: {}; indent: {}; number: {}'.format(args.width,
                                                       args.indent, args.number))
    if args.filename:
        print ('filename: ', args.filename)
    else:
        print ('no filename specified')

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

Another detour: "main" function

```
#!/usr/bin/env python # Must be where Python is located...

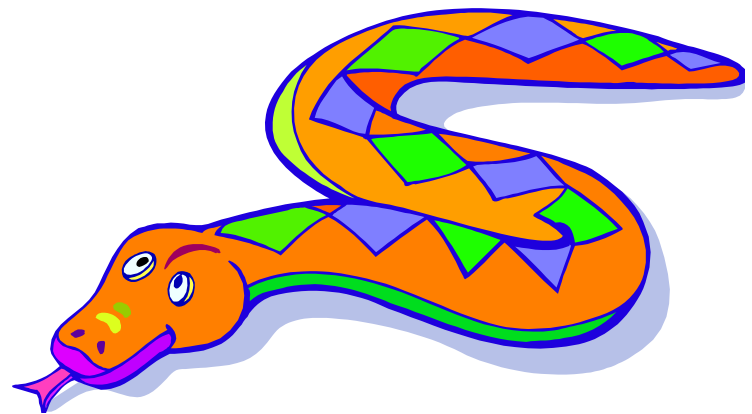
def main():
    print ("Here we are in main. About to visit caveOfCaerbannog.")
    caveOfCaerbannog()
    print()
    print ("Now we're back in main. About to call camelot().")
    camelot()
    print()
    print ("I feel happy! I feel hap...")

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 is a silly place.")

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.**

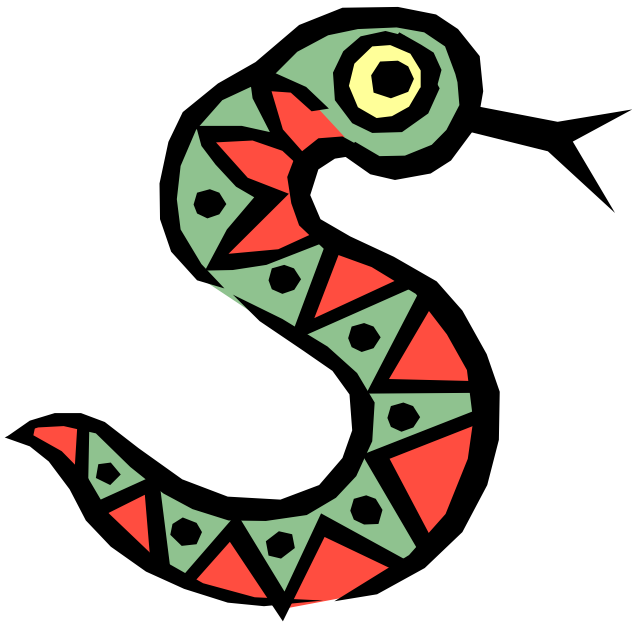
Special Properties of *and* and *or*

- 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)`

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*

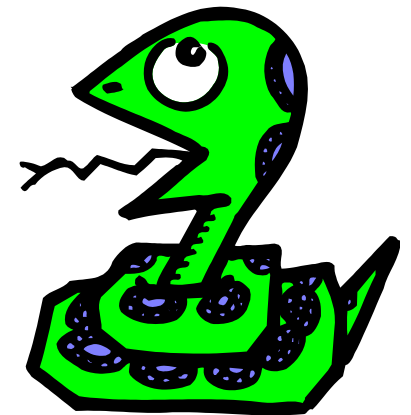
- You can use the keyword *break* inside a loop to leave the *while* loop entirely.
- You can use the keyword *continue* inside a loop to stop processing the current iteration of the loop and to immediately go on to the next one.

assert

- An *assert* statement will check to make sure that some condition is true during the course of a program.
 - If the condition is 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

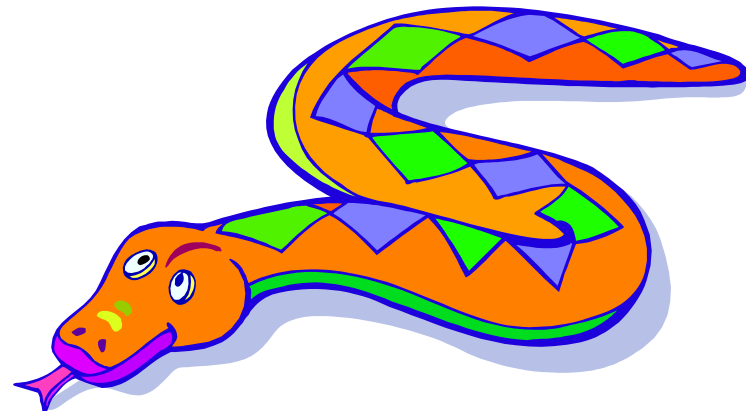
- Use `enumerate()`
- Returns a sequence of integer, item pairs
- Example

```
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.

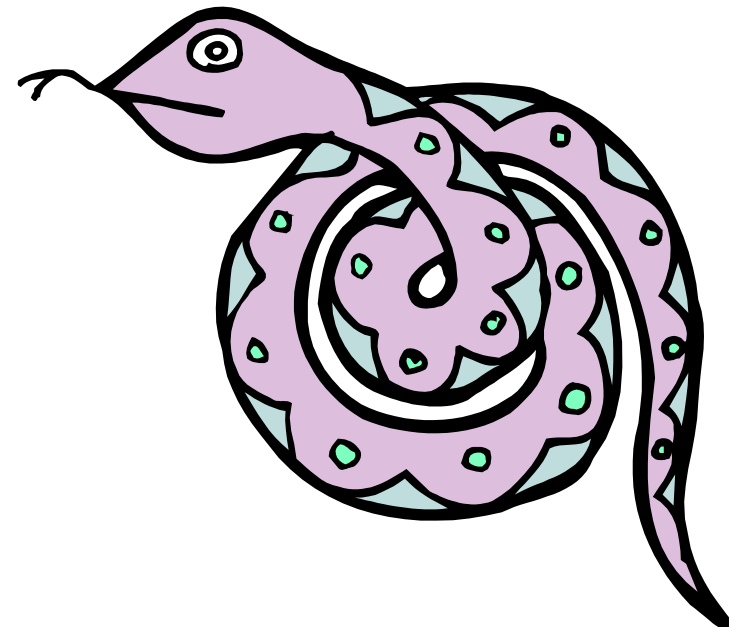
Nested List Comprehensions

- 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



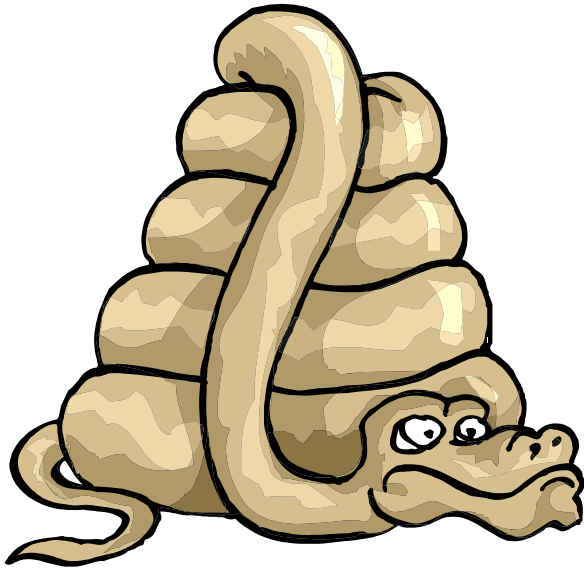
Lambda Notation

- 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: SEN; Number 265
```

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

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

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

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
```

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

```
    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:]
```

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

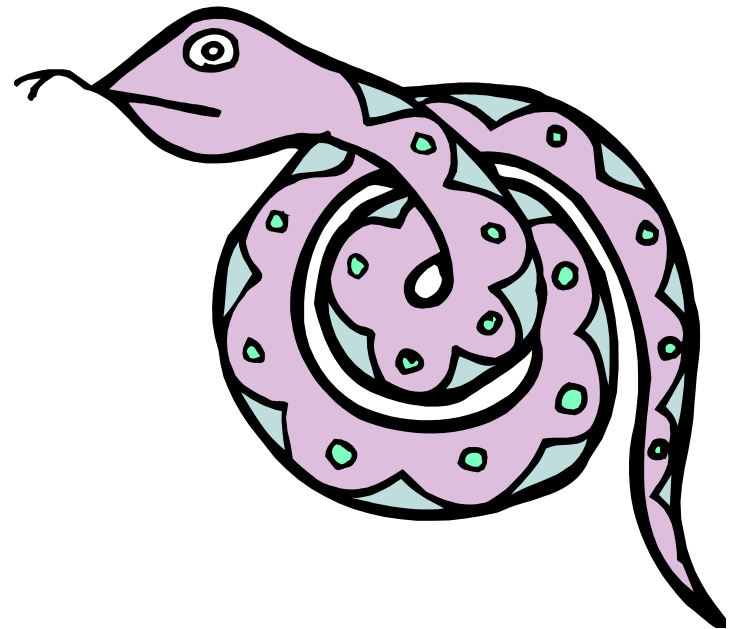
and accessing the tail of a list...

```
    print (num_lines, num_words, num_chars)
```

```
if __name__ == "__main__":
```

```
    main()
```

File Processing and Error Handling: Learning on your own...



File Processing with Python

This is a good way to play with the error handling capabilities of Python. Try accessing files without permissions or with non-existent names, etc.

You'll get plenty of errors to look at and play with!

```
import sys
fileptr = open('filename', 'r')
if fileptr == None:
    print("Something bad happened")
    sys.exit(1)
somestring = fileptr.read() # read one line
for line in fileptr:        # continue reading
    print (line)
fileptr.close()
```

Working with files

- **When opening a file, can specify whether:**

- We want to read or write content
- We want to treat content as strings or as byte arrays

```
file = open("somefile", "r")    # read strings
file = open("somefile", "rb")   # read binary arrays
file = open("somefile", "w")    # write strings
file = open("somefile", "wb")   # write binary arrays
```

- **The choice has important consequences**

- Regardless of how we process data in our Python code, we must be aware of how the file expects to read and write our data

Working with files

```
>>> file = open("fubar", "w")
>>> file.write("hamberders\n")
11
>>> file.close()

>>> file = open("fubar.bin", "wb")
>>> file.write("hamberders")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' does not support the buffer interface
```

```
$ hexdump -C fubar
00000000  68 61 6d 62 65 72 64 65  72 73 0a                |hamberders.|
0000000b
```

Writing specific ASCII values

- **This can be a bit tricky**
 - python supports both old-school ASCII (i.e., "latin-1" encodings) and more international UTF-8
 - Sometimes need to specify the encoding we want
 - Do not depend upon your platform defaults!

```
>>> file = open("oops.txt", "w")
>>> file.write("\xab\xba\xfe\xed")
4
>>> file.close()
```

```
$ hexdump -C oops.txt
00000000  c2 ab c2 ba c3 be c3 ad          |.....|
00000008
```

Writing specific ASCII values

```
>>> file = open("ver1.txt", encoding="latin-1", mode="w")
>>> file.write("\xab\xba\xfe\xed Geburtstag")
15
>>> file.close()
```

```
$ hexdump -C ver1.txt
00000000  ab ba fe ed 20 47 65 62  fc 72 74 73 74 61 67      |???? Geb?rtstag|
0000000f
```

```
>>> file = open("ver2.txt", encoding="utf-8", mode="w")
>>> file.write("\xab\xba\xfe\xed Geburtstag")
15
>>> file.close()
```

```
$ hexdump -C ver2.txt
00000000  c2 ab c2 ba c3 be c3 ad  20 47 65 62 c3 bc 72 74  |«?þí Geburt|
00000010  73 74 61 67              |stag|
00000014
```


Working with binary

- **Sometimes we need the "binary" representation of some integer value**
 - We can do the conversion...
 - ... and use the result
- **It looks a little tricky**
 - There will be other ways of doing this, but here we want to convert the binary representation into something writable to a file

```
>>> import struct
>>> bytes = struct.pack("I", 121)      # 121 as a four-byte int
>>> chars = [chr(c) for c in bytes]    # each byte as a ASCII
>>> s = "".join(chars)
>>> s
'y\x00\x00\x00'
>>> somefile.write(s)
```

Working with binary

- And given some "binary" representation, we may want to get the original int

```
>>> import struct
...
>>> s = somefile.read(4)
>>> chars = list(s)
>>> ints = [ord(c) for c in chars]
>>> bytes = bytearray(ints)
>>> result = struct.unpack("I", bytes)
>>> result
(121,)
>>> result[0]
121
```

Options with file input

Sometimes we want to simply treat files as a sequence of bytes rather than as line of characters, but not used the packed form.

```
f = open("somefile.bin", encoding="latin-1", \
        mode ="r")

# Assuming open was successful
while True
    a_byte = f.read(1)    # read one char/byte
    if not a_byte:        # if byte == None
        break
    val = ord(a_byte)     # ASCII value of byte
    print(val)
f.close()
```

Exception Handling

- **Errors are a kind of object in Python.**
 - More specific kinds of errors are subclasses of the general Error class.
- **You use the following commands to interact with them:**
 - try
 - except
 - finally
 - catch

Exceptions and handlers

```
while True:
    try:
        x = int(input("Number, please! "))
        print ("The number was: ", x)
        break
    except ValueError:
        print ("Oops! That was not a valid number.")
        print ("Try again.")
        print ()
```

Exceptions and handlers

```
try:
    f = open("gift.txt", "r")
    # do file-open things here
    f.close()
except FileNotFoundError:
    print("Cannot open gift.txt")
    # do file-doesn't-exist stuff here
except PermissionError:
    print("Whatcha trying to do here with gift.txt, eh?")
    # do failure-due-to-permissions stuff here
```

Exceptions and handlers

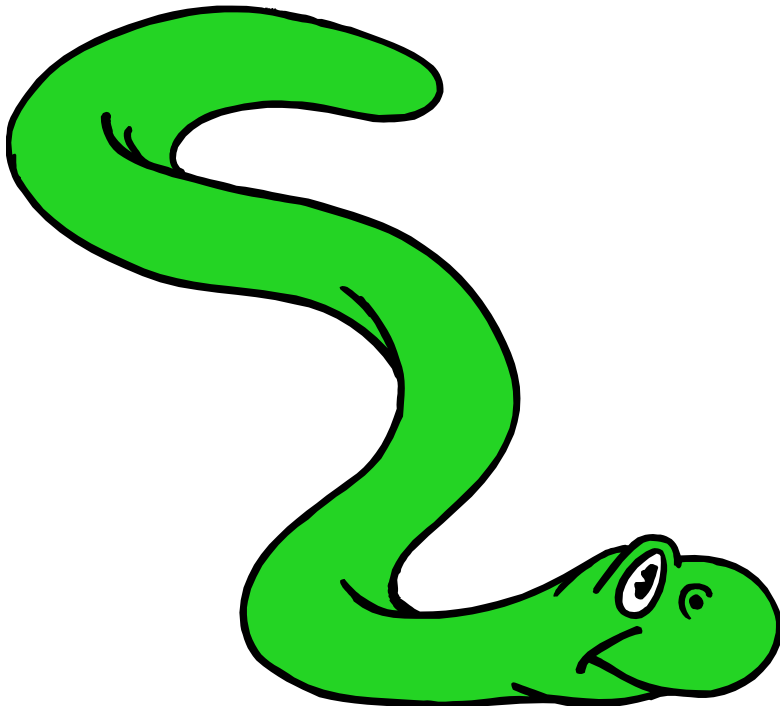
```
def loud_kaboom():
    x = 1/0;

def fireworks_factory():
    raise ZeroDivisionError("Gasoline near bone-dry Christmas trees!")

def playing_with_fire():
    try:
        loud_kaboom()
    except ZeroDivisionError as exc:
        print ("Handling run-time error: ", exc)

    try:
        fireworks_factory()
    except ZeroDivisionError:
        print ("Gotta stop this from happening...")
```

Scope Rules



Scope rules

- In Java or C: scope of variable depends upon (a) location of declaration and possibly (b) extra modifiers
- In Python: scope depends upon (a) location of variable definition and possible (b) the "global" modifier
- **LEGB rule used to resolve variable name:**
 - First search for Local definition...
 - ... and if not found there, go to Enclosing definition...
 - ... and if not found there go to "Global" definition...
 - ... and if not found there go to Built-in definition...
 - ... and if not found there, give up in despair.

Built-in (Python)

Names preassigned in the built-in names module: `open`, `range`, `SyntaxError`....

Global (module)

Names assigned at the top-level of a module file, or declared `global` in a `def` within the file.

Enclosing function locals

Names in the local scope of any and all enclosing functions (`def` or `lambda`), from inner to outer.

Local (function)

Names assigned in any way within a function (`def` or `lambda`), and not declared `global` in that function.

from Figure 17-1, "Learning Python, Fifth edition", O'Reilly (2013)

Scope examples

```
x = 22

def func():
    x = 33

func()
print(x)  # Prints 22: unchanged
```

```
x = 99

def func(Y):
    z = x + Y
    return z

func(1)    # result is 100
```

Scope examples

```
x = 88

def func():
    global x
    x = 99

func()
print(x)    # Prints 99
```

```
y, z = 1, 2

def all_global():
    global x
    x = y + z    # LEGB indicate show to interpret y and z
```

Nested function example (Python 3)

```
x = 99

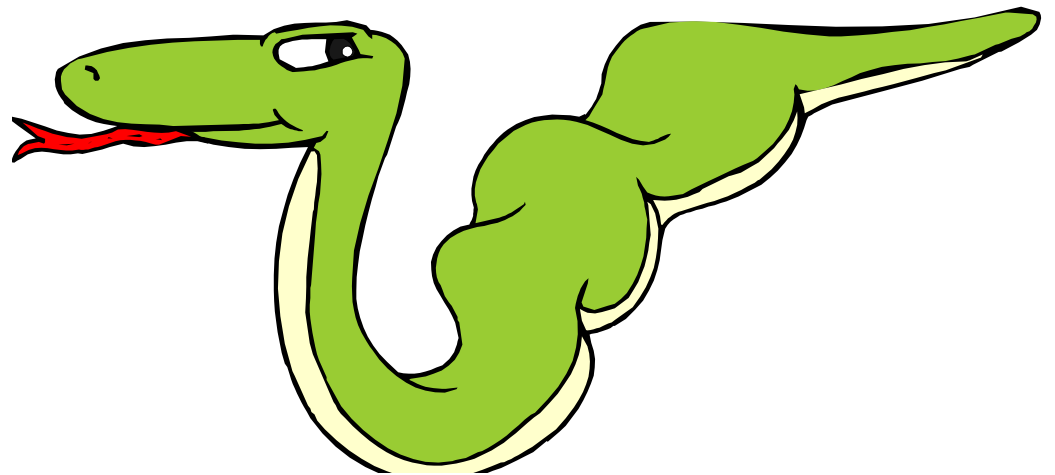
def f1():
    x = 88
    def f2():
        print(X)    # LEGB rule indicates via "E" the meaning of X
    f2()

f1()    # Value output: 88
```

Scope rules

- **"Global" scope is actually "Module" scope**
- **Usually a good idea to minimize use of global variables**
 - Although sometimes it does make sense for a module to have its own global variables
 - Need to practice some judgement here
- **Not covered here (more complex):**
 - class scopes
 - comprehension scope
 - closures

Assignment and Containers



Multiple Assignment with Sequences

- We've seen multiple assignment before:

```
>>> x, y = 2, 3
```

- But you can also do it with sequences.
 - The "shape" has to match.

```
>>> (x, y, (w, z)) = (2, 3, (4, 5))  
>>> [x, y] = [4, 5]
```


Empty Containers 1

- Assignment creates a name, if it didn't exist already.
`x = 3` Creates name `x` of type integer.
- Assignment is also what creates named references to containers.

```
>>> d = { 'a':3, 'b':4 }
```

- We can also create empty containers:

```
>>> li = []  
>>> tu = ()  
>>> di = {}
```

Note: an empty container is *logically* equivalent to `False`. (Just like `None`.)

- These three are empty, but of different *types*

Empty Containers 2

- **Why create a named reference to empty container?**

- To initialize an empty list, for example, before using append.
- This would cause an unknown name error a named reference to the right data type wasn't created first

```
>>> g.append(3)
```

Python complains here about the unknown name 'g'!

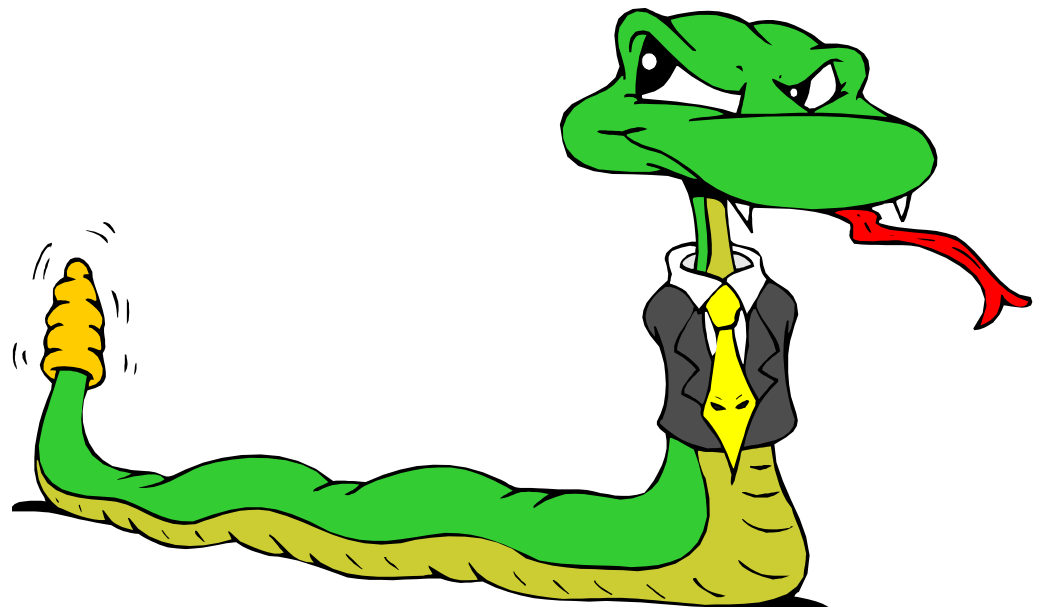
```
>>> g = []
```

```
>>> g.append(3)
```

```
>>> g
```

```
[3]
```

Importing and Modules



Importing and Modules

- Use classes & functions defined in another file.
- A Python module is a file with the same name (plus the *.py* extension)
- Like Java *import*, a little bit like C++ *include*.
- Three formats of the command:

```
import somefile
```

```
from somefile import *
```

```
from somefile import className
```

The difference? ?

What it is that is imported from the file and how we refer to the items after import.

import ...

```
import somefile
```

- *Everything* in somefile.py gets imported.
- To refer to something in the file, append the text "somefile." to the front of its name:

```
somefile.className.method("abc")
```

```
somefile.myFunction(34)
```

*from ... import **

```
from somefile import *
```

- ***Everything*** in somefile.py gets imported
- To refer to anything in the module, just use its name. Everything in the module is now in the current namespace.
- ***Caveat!*** Using this import command can easily overwrite the definition of an existing function or variable!

```
className.method("abc")
```

```
myFunction(34)
```

from ... import ...

```
from somefile import className
```

- Only the item *className* in somefile.py gets imported.
- After importing *className*, you can just use it without a module prefix. It's brought into the current namespace.
- **Caveat!** This will overwrite the definition of this particular name if it is already defined in the current namespace!

```
className.method("abc")  
myFunction(34)
```

← This was imported by the command.

← This one wasn't!

Commonly Used Modules

- **Some useful modules to import, included with Python:**
- **Module: sys**
 - sys.maxint
 - sys.argv
 - Lots of handy stuff.
- **Module: os**
 - OS specific code.
- **Module: os.path**
 - Directory processing.

More Commonly Used Modules

- **Module: math** - Mathematical code.
 - Exponents
 - sqrt
- **Module: Random** - Random number code.
 - Randrange
 - Uniform
 - Choice
 - Shuffle

Defining your own modules

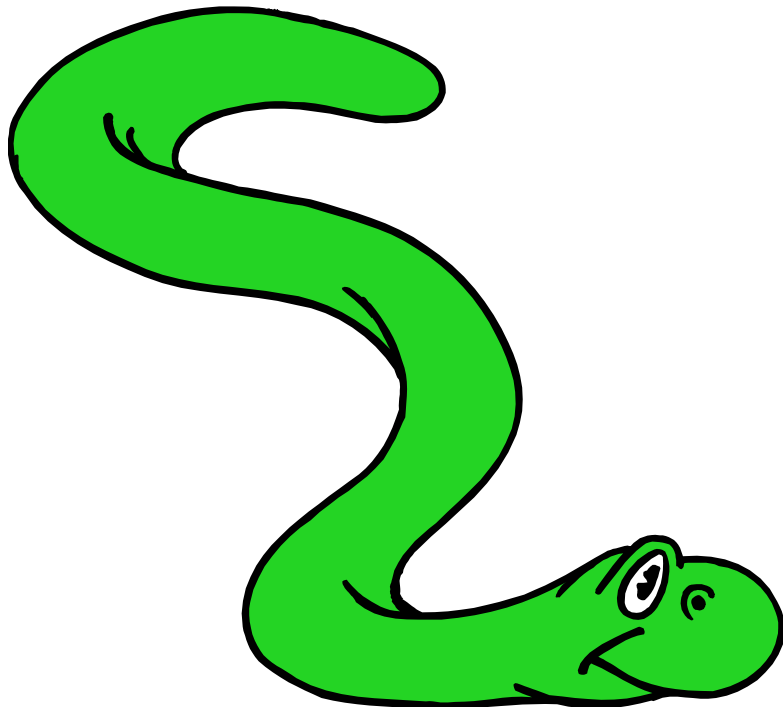
- You can save your own code files (modules) and import them into Python.

Directories for module files

Where does Python look for module files?

- The list of directories in which Python will look for the files to be imported: `sys.path`
(Variable named 'path' stored inside the 'sys' module.)
- To add a directory of your own to this list, append it to this list via a statement in your script.
`sys.path.append('/my/new/path')`

Object Oriented Programming in Python: Defining Classes



It's all objects...

- **Everything in Python is really an object.**
 - We've seen hints of this already...

```
"hello".upper()  
list3.append('a')  
dict2.keys()
```
 - These look like Java or C++ method calls.
 - New object classes can easily be defined in addition to these built-in data-types.
- **In fact, programming in Python is typically done in an object-oriented fashion.**

Defining a Class

- A **class** is a special data type which defines how to build a certain kind of object.
 - The **class** also stores some data items that are shared by all the instances of this class.
 - **Instances** are objects that are created which follow the definition given inside the class.
- **Python doesn't use separate class interface definitions as in some languages.**
 - You just define the class in the Python program...
 - ... and then use it.

Methods in Classes

- Define a *method* in a *class* by including function definitions within the scope of the class block.
 - There must be a special first argument *self* in all method definitions which gets bound to the calling instance
 - There is usually a special method called *__init__* in most classes
 - We'll talk about both later...

A simple class definition: *student*

```
class Student:
    """A class representing a student."""
    def __init__(self, n, a):
        self.full_name = n
        self.age = a
    def get_age(self):
        return self.age
```

Creating and Deleting Instances

Instantiating Objects

- There is no "new" keyword (i.e. Python is not the same syntactically as Java).
- Merely use the class name with () notation and assign the result to a variable.
- `__init__` function serves as a constructor for the class. Usually does some initialization work (of course).
- The arguments passed to the class name are given to its `__init__()` method.
 - So, the `__init__` method for `student` is passed "Bob" and 21 here and the new class instance is bound to `b`:

```
b = Student("Bob", 21)
```

Constructor: `__init__`

- An `__init__` method can take any number of arguments.
 - Like other functions or methods, the arguments can be defined with default values, making them optional to the caller.
- However, the first argument `self` in the definition of `__init__` is special...

self

- **The first argument of every method is a reference to the current instance of the class.**
 - By convention, we name this argument *self*.
 - We could give it a different name, but we'd risk writing unreadable Python code...
- **In `__init__`, *self* refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called.**
 - Similar to the keyword *this* in Java or C++.
 - But Python uses *self* far more often than Java uses *this*.

self

- Although you must specify *self* explicitly when defining the method, **you don't include it** when calling the method.
- Python passes it for you automatically.

Defining a method:

(this code inside a class definition.)

```
def set_age(self, num):  
    self.age = num
```

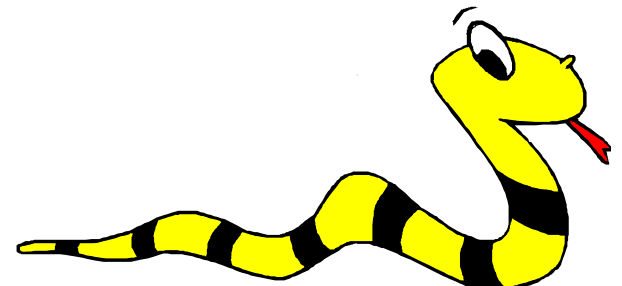
Calling a method:

```
>>> x.set_age(23)
```

Deleting instances: No Need to "free"

- **When you are done with an object, you don't have to delete or free it explicitly.**
 - Python has automatic garbage collection.
 - Python will automatically detect when all of the references to a piece of memory have gone out of scope. Automatically frees that memory.
 - Generally works well, few memory leaks.
 - There's also no "destructor" method for classes.

Access to Attributes and Methods



Definition of student

```
class Student:
    """A class representing a student."""
    def __init__(self, n, a):
        self.full_name = n
        self.age = a
    def get_age(self):
        return self.age
```


Traditional Syntax for Access

```
>>> f = Student ("Bob Smith", 23)

>>> f.full_name      # Access an attribute.
"Bob Smith"

>>> f.get_age()      # Access a method.
23
```

Accessing unknown members

- **Problem:** Occasionally the name of an attribute or method of a class is only given at run time...
- **Solution:** `getattr(object_instance, string)`
 - `string` is a string which contains the name of an attribute or method of a class
 - `getattr(object_instance, string)` returns a reference to that attribute or method
- **Only need this when writing very extensible code**

getattr(object_instance, string)

```
>>> f = Student("Bob Smith", 23)

>>> getattr(f, "full_name")
"Bob Smith"

>>> getattr(f, "get_age")
<method get_age of class studentClass at 010B3C2>

>>> getattr(f, "get_age")()    # We can call this.
23

>>> getattr(f, "get_birthday")
# Raises AttributeError - No method exists.
```

hasattr(object_instance,string)

```
>>> f = Student("Bob Smith", 23)
```

```
>>> hasattr(f, "full_name")
```

```
True
```

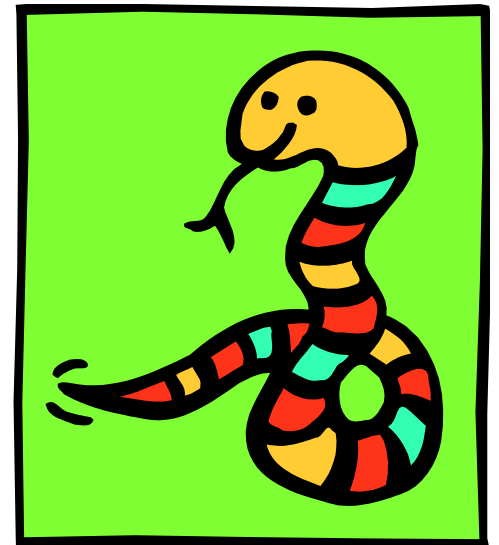
```
>>> hasattr(f, "get_age")
```

```
True
```

```
>>> hasattr(f, "get_birthday")
```

```
False
```

Attributes



Two Kinds of Attributes

- The non-method data stored by objects are called attributes.
- *Data* attributes
 - Variable owned by a *particular instance* of a class.
 - Each instance has its own value for it.
 - These are the most common kind of attribute.
- *Class* attributes
 - Owned by the *class as a whole*.
 - *All instances of the class share the same value for it.*
 - Called "static" variables in some languages.
 - Good for
 - class-wide constants
 - building counter of how many instances of the class have been made

Data Attributes

- Data attributes are created and initialized by an `__init__()` method.
 - Simply assigning to a name creates the attribute.
 - Inside the class, refer to data attributes using `self` for example, `self.full_name`

```
class Teacher:
    "A class representing teachers."
    def __init__(self,n):
        self.full_name = n
    def print_name(self):
        print(self.full_name)
```

Class Attributes

- **Because all instances of a class share one copy of a class attribute:**
 - when *any* instance changes it, the value is changed for *all* instances.
- **Class attributes are defined**
 - *within* a class definition
 - *outside* of any method
- **Since there is one of these attributes *per class* and not one *per instance*, they are accessed using a different notation:**
 - Access class attributes using `self.__class__.name` notation.

```
class Sample:
    x = 23
    def increment(self):
        self.__class__.x += 1
```

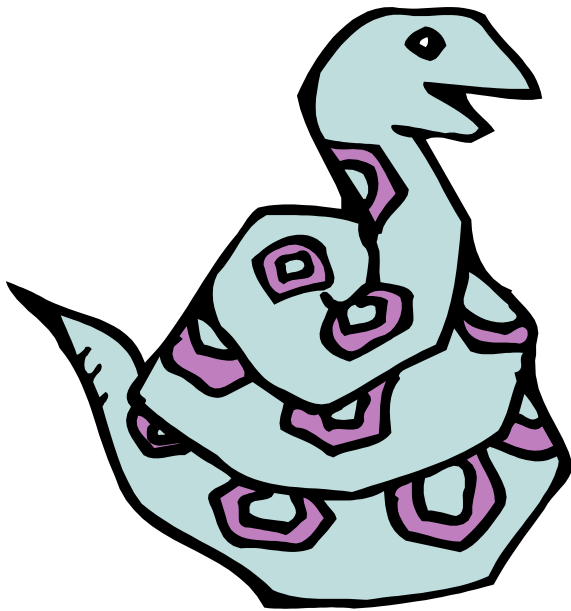
```
>>> a = Sample()
>>> a.increment()
>>> a.__class__.x
24
```


Data vs. Class Attributes

```
class counter:
    overall_total = 0
    # class attribute
    def __init__(self):
        self.my_total = 0
        # data attribute
    def increment(self):
        counter.overall_total = \
            counter.overall_total + 1
        self.my_total = \
            self.my_total + 1
```

```
>>> a = counter()
>>> b = counter()
>>> a.increment()
>>> b.increment()
>>> b.increment()
>>> a.my_total
1
>>> a.__class__.overall_total
3
>>> b.my_total
2
>>> b.__class__.overall_total
3
```

Inheritance



Subclasses

- A class can **extend** the definition of another class
 - Allows use (or extension) of methods and attributes already defined in the previous one.
 - New class: *subclass*. Original: *parent*, *ancestor* or *superclass*
- To define a subclass, put the name of the superclass in parentheses after the subclass's name on the first line of the definition.
`class AI_Student(Student) :`
 - Python has no 'extends' keyword like Java.
 - Multiple inheritance is supported.

Redefining Methods

- To *redefine a method* of the parent class, include a new definition using the same name in the subclass.
 - The old code won't get executed.
- To execute the method in the parent class *in addition to* new code for some method, explicitly call the parent's version of the method.

`ParentClass.methodName(self, a, b, c)`

- **The only time you ever explicitly pass 'self' as an argument is when calling a method of an ancestor.**

Definition of a class extending student

```
class Student:  
    "A class representing a student."
```

```
    def __init__(self, n, a):  
        self.full_name = n  
        self.age = a
```

```
    def get_age(self):  
        return self.age
```

```
-----  
class AI_Student (Student):  
    "A class extending Student."
```

```
    def __init__(self,n,a,s):  
        Student.__init__(self,n,a) #Call __init__ for student  
        self.section_num = s
```

```
    def get_age(self):          #Redefines get_age method entirely  
        print("Age: " + str(self.age))
```

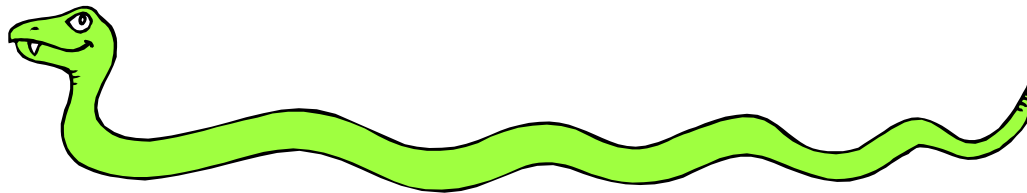
Extending `__init__`

- **Same as for redefining any other method...**
 - Commonly, the ancestor's `__init__` method is executed in addition to new commands.
 - You'll often see something like this in the `__init__` method of subclasses:

```
parentClass.__init__(self, x, y)
```

where `parentClass` is the name of the parent's class.

Special Built-In Methods and Attributes



Built-In Members of Classes

- **Classes contain many methods and attributes that are included by Python even if you don't define them explicitly.**
 - Most of these methods define automatic functionality triggered by special operators or usage of that class.
 - The built-in attributes define information that must be stored for all classes.
- **All built-in members have double underscores around their names: `__init__` `__doc__`**

Special Methods

- For example, the method `__repr__` exists for all classes, and you can always redefine it.
- The definition of this method specifies how to turn an instance of the class into a string.
 - `print(f)` sometimes calls `f.__repr__()` to produce a string for object `f`.
 - If you type `f` at the prompt and hit ENTER, then you are also calling `__repr__` to determine what to display to the user as output.

Special Methods – Example

```
class Student:
    def __str__(self):
        return "I am %s aged %d" % (self.full_name, self.age)

    def __repr__(self):
        return "Student(%s, %d)" % \
            (repr(self.full_name), repr(self.age))
    ...

>>> f = Student("Bob Smith", 23)
>>> print(f)
I am Bob Smith aged 23
>>> repr(f)
Student("Bob Smith", 23)
```

Special Methods

- **You can redefine these as well:**
 - `__init__` : The constructor for the class.
 - `__cmp__` : Define how `==` works for class.
 - `__len__` : Define how `len (obj)` works.
 - `__copy__` : Define how to copy a class.
- **Other built-in methods allow you to give a class the ability to use `[]` notation like an array or `()` notation like a function call.**

Special Data Items

- **These attributes exist for all classes.**

`__doc__` : Variable storing the documentation string for that class.

`__class__` : Variable which gives you a reference to the class from any instance of it.

`__module__` : Variable which gives you a reference to the module in which the particular class is defined.

- **Useful:**

- `dir(x)` returns a list of all methods and attributes defined for object `x`

Special Data Items – Example

```
>>> f = Student("Bob Smith", 23)

>>> print f.__doc__
A class representing a student.

>>> f.__class__
< class studentClass at 010B4C6 >

>>> g = f.__class__("Tom Jones", 34)
```

Private Data and Methods

- **Any attribute or method with two leading underscores in its name (but none at the end) is meant to be treated as private. It is not meant to be accessed outside of that class.**
 - Note:
Names with two underscores at the beginning *and the end* are for built-in methods or attributes for the class.
 - Note:
There is no 'protected' status in Python; so, subclasses would be unable to access these private data either.
 - Lastly:
Unlike Java, this naming convention makes access of such attributes more difficult, but does not entirely hide them from users of the object (i.e., there is no strict implementation of "private" in Python)

Private-ish

```
class Student:
    """A class representing a student, but where the
    attributes are named in a private-ish kind of way."""
    def __init__(self, name, aage):
        self.__full_name = name
        self.__age = aage

    def get_age(self):
        return self.__age

    def __str__(self):
        return self.__full_name + " (" + str(self.__age) + ")"
```

```
>>> s1 = Student("Bob Smith", 23)

>>> print(s1)
Bob Smith (19)

>>> print(s1.__full_name)
<... snip ...>
AttributeError: 'Student' object has no attribute '__full_name'
```

Private-ish

```
class Student:
    """A class representing a student, but where the
    attributes are named in a private-ish kind of way."""
    def __init__(self, name, aage):
        self.__full_name = name
        self.__age = aage

    def get_age(self):
        return self.__age

    def __str__(self):
        return self.__full_name + " (" + str(self.__age) + ")"
```

```
>>> print(s1.__age)
<... snip ...>
AttributeError: 'Student' object has no attribute '__age'

>>> dir(s1)
['_Student__age', '_Student__full_name', '__class__', '__delattr__', ...

>>> print(s1._Student__age)      # :-(
```


Classes seem kinda clunky, but...

- This would be to miss the main point
- That is:
 - Python does **support** the class mechanism...
 - ... but more importantly **it also supports** a framework of classes and types ...
 - ... plus a pattern of typical object usage ...
 - ... where what matters most is that operations are supported on class/type instances under that typical usage.
 - This is sometimes called “duck typing”