Artificial Intelligence Programming

Python

Cindi

Department of Computer Science

University of San Francisco

What is Python?

- Python is:
  - High-level
  - Interpreted
  - Object-oriented
  - Free, open-source
  - Dynamically typed
  - Has a large collection of utility libraries
  - garbage-collected
  - Mixable - works nicely as a “glue” language
  - Easy to learn/use

Some Uses of Python

- Things that Python is good at:
  - System utilities
    - Python has hooks into standard OS-level services, such as files, processes, pipes, etc.
  - Pattern recognition
  - GUIs
    - Python has interfaces to Tk, Qt, and WxPython
  - Embedded integration
    - Python has hooks that allow it to call/be called by C and C++ code, and also COM interfaces.

Some Uses of Python

- Rapid Interactive Development
  - Like Lisp, Python makes it easy to quickly put together a working program.
- Scripting
  - Python has utilities for CGI, Database access, HTTP, Sockets, FTP, POP/SMTP, XML parsing, etc.

Invoking Python Programs

- There are four ways to run a Python program:
  - Interactively
  - From the command line
  - As a script
  - From another program

Python Implementations

- /usr/bin/python on all unix machines and on OS X
- IDLE and PythonWin on Windows
- MacPython on Macintosh
- Also for Amiga, PalmOS, BeOS, etc.
  - USF has a license for PyCharm, runs on most systems, see Canvas References
  - Eclipse has a Python plugin (PyDev)
- We'll be using Python 2.7 in class.

Python Program Structure

- Programs consist of modules
  - A module corresponds to a source file.
- Modules contain blocks of statements
- Statements create and process objects.

Basic Python

- Python has a nice set of built-in data types
  - Numbers
  - Strings
  - Lists
  - Sets
  - Dictionaries
  - Files
- Using built-in types makes your code easier to write/maintain, more portable, and more efficient.

Numbers

- Numbers work like we'd expect.
- There are integers, equivalent to longs in C. (1, -31311, 4000)
- There are *long integers*, which are of unlimited size (31111L, 12345l)
- There are floats, equivalent to doubles in C or Java. 1.23, 3.1e+5
- There are Octal and Hexadecimal representations as in C. (0155, 0x3af5)
- There are complex numbers, as in Lisp, Matlab, etc. (3.0+4j)

Mathematical Operators

- Python has all the usual mathematical operators
  - +,=,*,/, %
  - «, »
  - ** or pow for exponentiation
  - abs, rand, |, &
- This makes Python a very handy desk calculator.
- Operations are coerced to the most specific type.
  - 3 + (4.0 / 2) = ?
  - Common error: since variables do not have declared types, be careful of rounding: 3 /2 = ?

Strings

- One of Python's strong suits is its ability to work with strings.
- Strings are denoted with double quotes, as in C, or single quotes
  - s1 + s2 - concatenation
  - s1 * 3 - repetition
  - s1[i] - indexing, s1[i:j] - slicing
  - s1[-1] - last character
  - "a % parrot" % "dead" - formatting
  - for char in s1 - iteration

Strings

- Access individual elements using subscripts:

```
>>> x = "Hello There"
>>> x[3]
'l'
```

  Note: not a char, but a string of len 1

- Access substrings using slices:

```
>>> x[3:5]
```

Strings

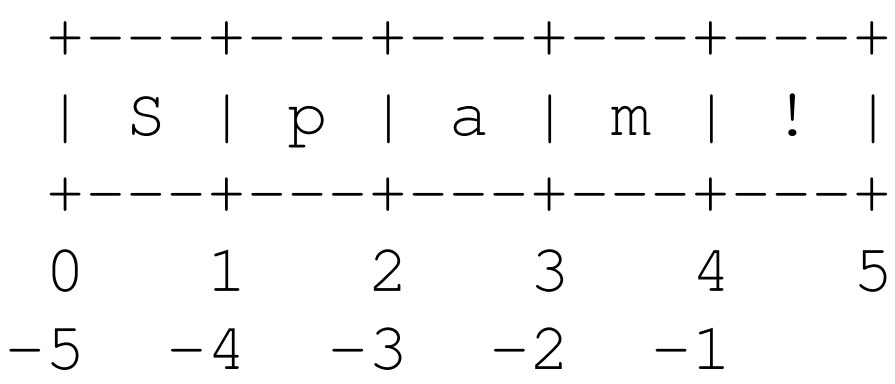
- Negative indices in slices count from the end of the string:

```
>>> x = "Hello There"
>>> x[0:-3]
'Hello The'
```

- Think of the indices as pointing between characters:

```
 +---+---+---+---+---+
 | S | p | a | m | ! |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

Strings



- What should this return?

```
>>> x = "Hello There"
>>> x[-5:-1]
```

Strings

- Strings are immutable sequences - to change them, we need to make a copy.
  - Can't do: s1[3] = 'c'
  - Must do: s2 = s1[0:2] + 'c' + s1[3:]
- As in Java, making lots of copies can be very inefficient. If you need to do lots of concatenation, use join instead.
- We'll return to efficiency issues throughout the semester.

Lists

- Python has a flexible and powerful list structure.
- Lists are mutable sequences - can be changed in place.
- Denoted with square brackets. l1 = [1,2,3,4]
- Can create nested sublists. l2 = [1,2, [3,4, [5], 6], 7]
  - l1 + l2 - concatenation.
  - l1 * 4 - repetition
  - l1[3:5], l1[:3], l1[5:] - slices
  - append, extend, sort, reverse built in.
  - Range - create a list of integers

Equality in Python

- Python is good at doing "what you want"
- "==" is value equality, not reference equality
- "is" is reference equality

```
>>> x = [1,2,3,4]
>>> y = [1,2,3,4]
>> z = x
>>> x == y
>>> x is y
>>> x is z
```

Dictionaries

- A Dictionary is a Python hash table (or associative list)
- Unordered collections of arbitrary objects.
- d1 = {} - new hashtable
- d2 = {'spam' : 2, 'eggs', 3}
- Can index by key: d2['spam']
- Keys can be any immutable object.
- Can have nested dictionaries
  - d3 = {'spam' : 1, 'other' :{'eggs' :2, 'spam' : 3}}
  - d3['other']['spam']

Dictionaries, cont'd

- Accessing and querying: has_key, keys(), values(), for k in keys()
- Typically, you'll insert/delete with:
  - d3['spam'] = 'delicious!'
  - del d3['spam']

Tuples

- Tuples are like immutable lists.
- Nice for dealing with enumerated types.
- Can be nested and indexed.
- t1 = (1,2,3), t2 = (1,2,(3,4,5))
- Can index, slice, length, just like lists.
  - t1[3], t1[1:2], t1[-2]
- Tuples are mostly useful when you want to have a list of a predetermined size/length.
- Also, constant-time access to elements. (fixed memory locations)
- Tuples are also very useful as keys for dictionaries.

Files

- Since it's a scripting language, Python has a lot of support for file I/O
- Operators are not too different from C.
- Outfile = open('fname', 'w') or infile = open('fname', 'r')
  - 'r' is default and can be left out
- S = infile.read() - read the entire file into the string S.
- S = infile.read(N) - read N lines into the string S.
- S = input.readline() - read one line
- S = input.readlines() - read the whole file into a list of strings.
  - Unless the file is *really* huge, it's fastest to read it all in at once with read() or readlines()

Files

- outfile.write(S) - write the string S into the file.
- outfile.writelines(L) - write the list of strings L into the file.
- outfile.close() (this is also done by the garbage collector)

Basic Python statements

- Python uses dynamic typing.
  - No need to pre-define variables.
- Variables are instantiated by assigning values to them
  - Referencing a variable before assignment is an error
- You can assign multiple variables simultaneously

```
spam = 4
eggs = 5
spam, eggs = eggs, spam
spam, eggs = 4,5
```

Python variables

- Variables must:
  - begin with a letter or underscore
  - contain any number (one or more) of letters, numbers, or underscores.
- No $, @, #, etc.
- Case matters.
- Can't use reserved words as variables.

Printing

- We've already seen the basic print.
  - print "hello world"
- To use a formatting string, do:
  - print "hello %s" % "bob"
  - print "%s %s" % ("hello" , "world")
- To suppress the linefeed, include a ,

Conditionals

- The general format for an if statement is:

```
if <test1>:
    <statement1>
    <statement2>
elif <test2>:
    <statement3>
...
else:
    <statementn>
```

- Notice the colons after the conditionals.
- Compound statements consist of the colon, followed by an indented block.

Booleans

- False
  - False - (built in, case sensitive!)
  - 0, 0.0 - (rounding can cause problems)
  - () - (empty tuple)
  - []- (empty list)
  - {} - (empty dictionary)
  - "" - (empty string)
- True
  - Anything else

Booleans

- a and b
- a or b
- not a

Syntax

- Indentation is used to delimit blocks
  - (If you are going to be editing your code on multiple machines with different editors, you may want to configure your editor to use spaces instead of tabs.)
  - Statements can cross multiple lines if:
    - They're within delimiters
    - You use a backslash

Iteration

- Python has the familiar while loop.
- One wrinkle: it has an optional else clause to execute if the test is false.
- Additional loop control
  - break
    - Exit the innermost loop without doing an else
  - continue
    - Return to the beginning of the loop
  - pass
    - Do nothing

For

- For is a generic iterator in Python.
- Lets you loop over any indexable data structure.
  - for item in [1,2,3,4,5]
  - for char in “This is a string”
  - for key in dictionary
- This provides a uniform (polymorphic) interface to a set of different data structures.

Efficiency of For

- Note: it's much faster to use the polymorphic operator than to access manually.
- Which is faster?

```
for item in list:
   print item
OR
for i in len(list):
   print list[i]

for item in dictionary:
   <do something with item>
OR
for item in dictionary.keys():
   <do something with item>
```

List comprehensions

An example of *functional programming*

Lets you *map* a function onto a sequence

[f(x) for x in L]

```
>>> [x*x for x in range(1,10)]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> z = ["hello", "there"]
>>> [y.upper() for y in z]
['HELLO', 'THERE']
```

List comprehensions with Tests

[f(x) for x in L if test]

```
>>> [x for x in range(1,10) if x % 2 == 0]
[2, 4, 6, 8]
>>> phoneBook = {"brooks": "",
  "parr": "422-3435", "thompson": "422-4017",
  "wolber": "422-1234"}
>>> ["name: %s phone %" %(nm ph)
     for nm, ph in phoneBook.items()
     if not ph==""]
```

Handy for conditional processing of lines in files

```
>>> data = [line.strip() for line in open(fname)
     if len(line) > 0]
```

List Comprehensions

They are not only useful, they're more efficient than traditional for loops

Allows the interpreter to optimize

Write your loop "inside out"

```
##example: filter a list of words to only find
## those that have four letters
final = []
for word in wordlist:
    if len(word) == 4:
        final.append(word)
```

Building the List comprehension

What result (list item) do we want to produce?

Building the List comprehension

What result do we want to produce?

- The word
- [word . . .]

What are we iterating over?

Building the List comprehension

What result do we want to produce?

- The word

What are we iterating over?

- wordlist
- [word for word in wordlist. . .]

In what conditions do we want to produce the word?

Building the List comprehension

What result do we want to produce?

- The word

What are we iterating over?

- wordlist

In what conditions do we want to produce the word?

- when it has 4 letters
- [word for word in wordlist if len(word) == 4]

Functions

- def is used to define a function

```
def <name>(params):
    <body>
```

- Params are all pass-by-value (like C/Java)
- "return <val>" returns a value, and is optional
- Functions maintain local scope
- Names are resolved locally, then globally, then built-in

Functions

- Multiple values can be returned in a tuple.
- We can also provide default arguments.
- Functions can be called with keywords for the args
  - myfunc(spam=1, eggs=2)
- *args can be used to catch arbitrary argument lists and store them in a tuple.
- **args can be used to catch arbitrary argument lists and store them in a dictionary.

Function comments

```
def <name>(params):
""" Comment that describes the function.
Please always use me!
"""
    <body>
```

- Comment an be accessed with help(functionname)

Functions as Objects

In Python, functions are first-class objects

- Can be assigned to variables, passed as args, evaluated
- Allows us to create higher-order functions

```
def cube(x): return x *  x * x
def my_map(list, fn):
    retVal = []
    for item in list:
        retVal.append(fn(item))
    return retVal
```

Basic Python advice

- “batteries included”
- Let the language do the work for you
  - e.g.: "if x in [1,2,3,4]" rather than an explicit loop
  - this is both faster and more readable
- Use a Python-aware editor/IDE
- Take advantage of the built-in modules whenever possible.
- Use the interpreter to help test your code
- All built-in modules are readable
  - Looking at these can give insights into how they work, as well as how to write good Python code.

Modules

- Each .py file is a "module"
- Can load "module.py" with "import module"
- Needs to be in PYTHONPATH env variable (or current dir)
- Other types of import
  - from <module> import <symbol>
  - from <module> import *

Module Gotchas

- When you import, code in file is executed
  - "def"'s generate functions
- Use of .py files
  - scripts: run from cmd line
  - modules: imported by other programs
- __name__ will have the value __main__ iff file is being used as a script

```
if __name__ == "__main__":
  <main program of script here>
```

Commonly used Python Modules

Some modules you'll use in this class:

- sys (mostly for argv)
- urllib and urllib2
- re (regular expressions are your friend)
- pickle/cPickle (serialize objects)
- time, datetime
- heapq (heap implementation of a priority queue)

Classes and Objects in Python

- Python has much of the same support for object-oriented programming as other languages, such as Java or C++.
- A few wrinkles:
  - multiple inheritance
  - no public/private
  - operator overloading

Python Classes

- Classes are defined with the 'class' keyword:

```
class Person:
  ...
```

- This defines a class with no parent class.

init

- __ init__ is the first method called when an object is created.
  - Technically, it's not a constructor, but close enough
- Takes at least one argument: self
- This is a pointer to the object

Example

```
class Person:
   def __init__(self, n):
      self.name = n
```

- This defines the person class. Persons have one instance variable, referred to as name.
- Must be scoped with the 'self' keyword.
- No distinction between declaration and assignment
- Common error: defining variables outside of a method. This creates a class variable

More on init

- Like all methods, init can used named arguments. This is how to implement multiple constructors.

```
class Person:
   def __init__(self, n=''bob'', height=72):
      self.name = n
      self.height=height
```

- This could be invoked with:
  - p = Person()
  - p = Person(“mary”, 66)
  - p = Person(height=76)
  - p = Person(n=”joe”)
  - p = Person(“joe”)

Methods

- Defined with def
- Inside the scope of the class definition.
- self is first argument in declaration
  - Not provided when calling the method - the Python interpreter fills this in.
- Can use default and keyword arguments

Example

```
class Person:
   def __init__(self, n=''bob'', height=72):
      self.name = n
      self.height=height

   def heightInCm(self):
      return self.height * 2.54

   def saluation(self, greeting):
      print greeting + '  ' + self.name
```

- Invoked as:
  - p.heightInCm()
  - p.salutation(“hello”)

Class variables

Declared outside the scope of a method

Referenced with the class name + var name

Useful for constants, counters, mutex/semaphores, etc.

Also for places where you don't want to create an object

- string.letters
- string.lowercase
- ...

Overloading operators

- Python classes have a variety of built-in operators that can be overloaded to produce specific behavior.
- These methods all begin and end with __
- This is a way of providing *polymorphism*
  - __repr__ - controls how an object is printed.
  - __lt__, __gt__, __le__, __gt__, __cmp__ : comparison operators
  - __add__, __sub__, __mul__, __div__ : arithmetic operators

Example

```
class Person :
   def __init__(self, n=''bob'', height=72):
       self.name = n
       self.height=height

  def __repr__(self):
       return 'my name is: ' + self.name
## assume we'll sort by height
  def __lt__(self, other):
       return self.height < other.height
```

- We can now do:
- p1 = Person('bob', 72)
- p2 = Person('joe', 70)
- if p1 < p2: ...

Overloading

- Overloading is sometimes a controversial subject
- Some language designers dislike them because they can be misued.
- Python takes the approach that you can have enough rope to hang yourself with.
- Be sensible - don't overload + to mean something unusual, for example.

Inheritance

- Like (almost) all OO languages, Python supports inheritance.
- Include the name of the parent class in parentheses.
- To call a parent class' method, use the name of the parent class.

Example

```
class Pet:
   def __init__(self, name="polly", species="parrot"):
      self.name = name
      self.species = species

    ...

class Cat(Pet):
   def __init(self, name, eats_parrots):
      Pet.__init__(self, name, "cat")
      self.eats_parrots = eats_parrots
```

Multiple Inheritance

- Python also supports multiple inheritance
- List multiple classes in declaration
- Names are resolved from left to right

Example

```
## professors teach classes
class Professor(Person): ...

## staff are administrators
class Staff(Person): ...

### but some staff also teach classes

class TeachingStaff(Professor, Staff): ...
```

Reminders

- Always need to use self to refer to member variables
- All variables are public
  - can use __spam to indicate private variables
- Classes can tell you what methods they implement with dir()
- this is called *introspection* (reflection in Java)
- Other introspection examples: type and obj.__doc__