Introduction to Python
Lecture II

Worcester Polytechnic Institute
Academic & Research Computing

Overview

What we will cover in this lecture

- Quick review of the first lecture
- Expand on lists
- Solution to some of the exercises
- Dictionaries
- Tuples
- Error Handling
- Files
- Moving to Anaconda
- Where to go from here?

Core Data Types

Python contains a variety of built-in object types for you to use.

Object Type	Example Literals/Creation
Numbers	`1234, 3.1415, Decimal(), Fraction()`
Strings	`'gompei', "Bob's"`
Lists	`[1, [2, 'three'], 4.5], list(range(10))`
Dictionaries	`{'mascot': 'gompei', 'animal': 'goat'}, dict(hours=10)`
Tuples	`(1, 'spam', 4, 'U'), tuple('goat'), namedtuple`
Files	`open('goats.txt'), open(r'C:\goat.bin', 'wb')`
Sets	`set('abc'), {'a','b','c'}`

Numbers

Numbers in Python are straightforward, and this **object type** contains the usual things you would expect:

- integers
- floats
- complex
- fixed precision decimals
- rationals
- sets

All of our number objects can be manipulated with the normal mathematical operators (+, -, *, **, /)

```
>>>3.1415 * 2
6.283
```

In addition to the basic mathematical operators, Python includes useful numeric modules (*a package of tools*) that can be imported.

Strings

Strings in Python have a much more rich set of uses and behaviors. Strings are the first example of a *sequence* in Python, or a:

"…positionally ordered collection of other objects."

What does this mean?

Strings are sequences that are positionally ordered, each item stored left to right, and can be fetched by their positions.

Technically, strings are sequences of single character strings.

Lists

Lists in Python are also generally sequences, just as strings are.

Lists are *positionally ordered* collections of *arbitrarily typed* objects.

Lists do not have a fixed size (remember, they are *mutable*).

Because of their mutability, we can modify lists in place, providing a very flexible tool for representing arbitrary collections.

Since lists are sequences just like strings, lists support all of the sequence operations we discussed for strings.

Polymorphism

The first important concept is that the plus sign (+) can mean different things for different objects. It means addition for numbers, but concatenation for strings.

We call this property ***polymorphism***, and it is a general property in Python. The ideas behind this are beyond the scope of our introduction to Python today, but you should be aware of it.

Immutability

Every object in Python is either immutable, or not.

Immutable objects are: numbers, strings, tuples

Mutable objects are: lists, dictionaries, sets

This consistency of certain objects can be used to guarantee that an object remains constant throughout your program.

Mutable objects could change at any time, by accident or intentionally, creating unexpected results if you aren't careful.

Again, the details are beyond the scope of this introduction, but ***immutability*** is an important concept you need to be aware of with Python.

Functions

A function is a device that groups a set of statements together so that they can be run as part of a more complicated program.

Functions are the most basic program structure in Python, and provide the best way to reuse code that you write.

Statement or expression	Example Literals/Creation
Call expressions	`myfunc('gompei', 42)`
def	`def printer(message)` `  print('Hello' + message)`
return	`def adder(a, b=1)` `  return a + b`

Functions


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

name = raw_input ("Input your name : ")
def namePrinter(name):
   greeting = "Hi, " + name
   return greeting

namePrinter(name)
```

We can also just loop over lists using `for`:

```
#!/usr/bin/env python
L = [1, 2, 3, 4, 5, 6]
sum = 0
for num in L:
   sum += num
print sum
```

You can see the iterative pattern over the list elements.

List Nesting

Lists in Python can also be nested, which can be used to represent matrices if you want.

In practice, you would use available packages like Numpy to do matrix operations, but still, you could do it if you wanted.

Either way, a nested list for any purpose can be indexed like this:

```
>>> M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> M[1]
[4, 5, 6]
>>> M[1][2]
6
```

Lists: Comprehensions

One of the more useful features in Python is *iteration*.

Sequences in Python are ***iterable***, meaning there is a sort of built-in mechanism for looping or iterating on lists.

A first example of this is ***list comprehensions***

```
>>> col2 = [row[1] for row in M]
>>> col2
[2, 5, 8]
```

You can see is that ***list comprehensions*** build new ***lists***.

Lists: Comprehensions

A for loop like this:

```
for (set of values to iterate):
  if (conditional filtering):
    output_expression()
```

Turns into an expression like this:

```
[output_expression() for(set of values to iterate) if(conditional filtering)]
```

Using our list matrix from before

```
>>> M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

col2 = []
for row in M:
 col2.append(row[1])

col2 = [row[1] for row in M]
```

Lists: Comprehensions

We can do more complicated things in list comprehensions though, like modify the new list as it is being built:

```
>>> colX2= [row[1] * 2 for row in M]
>>> colX2
[4, 10, 16]
```

Or we can filter out results we don't want:

```
>>> [row[1] for row in M if row[1] % 2 == 0]
[2, 8]
```

Things can get out of hand quickly:

```
>>> [[x, x/2, x*2] for x in range(-6, 7, 2) if x > 0]
[[2,1,4],[4,2,8],[6,3,12]]
```

Lists: Comprehensions

So the real question is, why would we do this to ourselves?

List comprehensions ***can be*** ~25% faster than for-loops. Not always true though!

List comprehensions can be more concise

List comprehensions are good for ***simple*** expressions

Practice!

Maths:

1. Write a program that, given a list of numbers, returns a list where all adjacent == elements have been reduced to a single element (e.g. [1, 2, 2, 3] returns [1, 2, 3])
2. Write a program that computes the factorial of a number
3. Write a program that calculates the sum of Fibonacci numbers below a threshold
4. Write a program that takes as input from the user the radius of a circle, and outputs the area of the circle.

Games:

1. Write a guessing game program, where the computer holds a random number between 1 and 100, and you get a certain number of tries to guess the number
2. Write a program that plays a simple version of hangman
3. Write a program that plays Battleship

Dictionaries

Dictionaries are:

- Accessed by key, not offset position
- Unordered collections of arbitrary objects
- Variable-length, heterogeneous, and arbitrarily nestable
- Of the category “mutable mapping”
  - You can change dictionaries in place by assigning to indexes, but they do not support the sequence operations that work on strings and lists (think slicing).
  - Because dictionaries are unordered collections, operations that depend on a fixed positional order don’t make any sense.

Dictionaries


```
>>>states = {
    'Minnesota': 'MN',
    'Iowa': 'IA',
    'Illinois': 'IL',
    'Massachusetts': 'MA',
}

>>>print states
{'Massachusetts': 'MA', 'Iowa': 'IA', 'Minnesota': 'MN',
'Illinois': 'IL'}
```

Dictionaries: Why?

Both dictionaries and lists are flexible containers for collections of objects

However, lists assign a *position* to the items it contains, whereas dictionaries assign *keys*

You need to think about your problem, and what type of data structure makes the most sense

Dictionaries tend to be best for data with labeled components, as well as structures that need quick, direct lookups by key.

Comparatively, it is much slower to search a list linearly than look up by key

Tuples

Tuples construct simple groups of objects

They function exactly like lists, however

Tuples are ***immutable***

You can think of tuples as immutable lists, with much of the same functionality and methods

Use parentheses to create a tuple, instead of the square brackets of lists

You probably noticed the tuples when we were talking about dictionaries

Error Handling: Testing Inputs and Using `try` Statements

We saw something similar to this before, when we were taking input from raw_input()

Sometimes you need to protect yourself from user input.

It is easiest to show you the two methods (testing inputs and the `try` statement) interactively

Files

File objects in Python are the main interface to external files on your computer. We can read and write to a huge variety of files, including text files, Excel spreadsheets, audio files, emails, etc, etc.

To create a file you call the built-in `open` function, passing in an external filename and an option 'mode' such as read or write.

```
>>> f = open('data.txt', 'w')
>>> f.write('Hello\n')
>>> f.write('world\n')
>>> f.close()
```

These commands create a file in your current directory and writes out the text to that file.

To read a file back in, open it with a `'r'` processing mode. The read mode is the default if you do not specify one.

To read a file back in, open it with a `'r'` processing mode. The read mode is the default if you do not specify one.

```
>>> f = open('data.txt')
>>> text = f.read()
>>> text
>>> 'Hello\nworld\n'
>>> print(text)
Hello
world
```

To read a file back in, open it with a `'r'` processing mode. The read mode is the default if you do not specify one.

```
>>> f = open('data.txt')
>>> text = f.read()
>>> text
>>> 'Hello\nworld\n'
>>> print(text)
Hello
world
>>> text.split()
['Hello', 'world']
```

Moving to Anaconda

On the workstations, open the Anaconda Prompt

This is the same as the standard Windows command prompt, but includes hooks into the Anaconda Python installation.

Once you have the command prompt open, simply type the command `python`

The command prompt will change from > to >>> after printing out some version information

This is the **interactive** Python shell

Here, you can set variables, print the variables, do basic arithmetic, etc.

Setting variables in Python is as easy as using an equal sign (=)

Printing variables in Python uses the `print` command

Moving to Anaconda

On the workstations, open Notepad++ (not Notepad)

From here, we are going to recreate out very first example, but instead save the commands inside of a Python script

Just as with bash or csh shell scripts, we include a line to tell the OS what kind of interpreter to run

At the top of your file, type:

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

Now put in your commands from the interactive prompt, such as setting the values of `mascot`, `a`, `b`, `f`, etc, as well as all the `print` statements (we actually need them now!)

Save the file to the desktop as `gompei.py`, and switch back to the Anaconda Prompt

Type `exit()` to leave the Python interpreter

Moving to Anaconda

From Anaconda Prompt, type:

```
python gompei.py
```

Instead of opening up a Python command prompt, the Python interpreter will now run through your file, executing the commands one at a time.

These two ways of using Python, interactive commands and file based programs, allow you to quickly test and debug commands, and then save them to a file as part of a program.

Now What?

From here, you just need to practice!

You won't break your computer (I think), so try things. You'll learn more from error messages than you will from getting things right.

If you need more problems, and like math, try Project Euler! It is a bottomless pit of math problems to be solved

For day to day stuff, if you need to do something more than 3 or 4 times, try scripting it! Writing simple one off scripts is a good way to make your life easier, and get some practice writing code as well.