PC204 Lecture 6

Conrad Huang
conrad@cgl.ucsf.edu
Genentech Hall, N453A
x60415

Topics

- Homework review
- Exceptions
- Module syntax
- Using modules
- Class syntax and using classes

Homework Review

- 5.1 – count_extensions
- 5.2 – get_pdb_ligands

Exception Handling

- Exceptions are “thrown” when Python encounters something it cannot handle
- Exceptions can indicate one of several conditions
  - Error in code
  - Error in execution, eg bad data
    - Fatal
    - Recoverable
  - Expected (usually rare) execution state

Coding Errors

- Some exceptions are generated by faulty code that never work correctly

```
>>> data = "this is not an integer"
>>> "%d" % data
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: int argument required
>>> "%d %d" % (1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: not all arguments converted during string formatting
```

- Solution: Fix the code

Execution Errors

- Some exceptions are generated by code that work some of the time
  - For example, this code might throw an exception if an expected data file is missing

```
>>> f = open('bee.tsv')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IOError: [Errno 2] No such file or directory: 'bee.tsv'
```

- Solution: ?

Recoverable Execution Errors

- Some exceptions need not be fatal
  - For example, if our expected TSV file is missing, we can regenerate it from the original text
- To recover from an exception, we have to “catch” the thrown exception by placing the suspect code inside a “try” statement

try .. except .. else

- The general form of a try statement is:

try:
    suspect statements
except *exception_type1* **[**, *exception_data1***]**:
    recovery statements 1
except *exception_type2* **[**, *exception_data2***]**:
    recovery statements 2
else:
    normal completion statements
finally:
    always executed whether exception thrown or not

- There must be at least one **except** clause
- The **else** and **finally** clauses are optional

try Statements

- A try statement is executed as follows:
  - "suspect statements" in the **try** clause are first executed
  - If they do not generate an exception, "normal completion statements" in the **else** clause are executed
  - Otherwise, the thrown exception is matched against the "exception_type"s listed in the **except** clauses and the corresponding "recovery statements" are executed
  - If an exception is thrown but no matching except clause was found, the exception is handled "normally" (as if the suspect statements were not in a try statement) and neither normal completion statements nor any recovery statements are executed

try Statements

- Note that the statements inside the **try** may be function calls
  - The called function may also have **try** statements
  - When an exception is raised, the last function called has the first opportunity to catch it
  - If a function does not catch an exception, the “stack is unwound” and its caller gets a chance to catch the exception
  - This continues until the main program has a chance to catch the exception
  - Finally, the Python interpreter catches and reports any uncaught exceptions

try Statements

- Statements in the **finally** clause are always executed
  - If no exception is thrown and there is an **else** clause, the statements are executed after the **else** clause
  - If no exception is thrown and there is no **else** clause, the statements are executed after the **try** statements
  - If an exception is thrown and is caught by an **except** clause, the statements are executed after the **except** clause
  - If an exception is thrown and is not caught, the exception is temporarily caught, the **finally** statements executed, and the exception rethrown

try Statement (cont.)

- Example of recovering from missing data file:

```
try:
        f = open("bee.tsv")
except IOError:
        f = open("bee.txt")
        # Regenerate data directly from bee.txt
        f.close()
else:
        # Read cached data from bee.tsv
        f.close()
```

- Note that if bee.txt is also missing, an IOError exception will still be thrown
  - There is no **try** in the recovery statements

Other Uses of **try** Statements

- **try** statements can be used deliberately in anticipation of rare or unusual conditions:

```
# Suppose d is a dictionary and we do different things depending
# on whether my_key appears as a key in d
# Approach A: LBYL - Look Before You Leap
if my_key in d:
        # Do something with d[my_key]
else:
        # Do something else

# Approach B: EAFP - Easier to Ask for Forgiveness than Permission
try:
        # Do something with d[my_key]
except KeyError:
        # Do something else
```

LBYL *vs* EAFP

- EAFP is endorsed by many Python experts because it tends to be more efficient and the code is generally easier to read
  - There are fewer tests being performed
  - The unusual conditions are distinctly and explicitly separated from the normal execution flow

Pitfalls of **try** Statements

- It is possible to use a bare "**except:**" clause (without specifying exception types) in a **try** statement
  - It is tempting to use this because it enables our programs to continue executing in the presence of errors
  - Unless we plan to handle all types of exceptions, this is a bad idea because it tends to intercept errors from any "higher level" **try** statements that may properly recover from some types of errors

Writing Modules

- Although Think Python only spends one page on "Writing Modules", there is actually quite a bit more to say
- Syntax for using multiple modules in a single program is very straightforward
- Reasons for using modules and how code should be organized is more complex
  - Avoid code duplication in multiple programs
  - Help organize related functions and data

Module Syntax

- Python treats any file with the .py suffix as a module, with the caveat that the part of the file name preceding .py consists of only legal Python identifier characters
- For example, wc.py

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print linecount("wc.py")
```

Module Syntax (cont.)

- To use the wc module, we need to import it

```
>>> import wc
7
>>> print wc
<module 'wc' from 'wc.py'>
>>> import wc
>>> wc.linecount("wc.py")
7
>>> wc.linecount("bee.tsv")
75
```

Module Syntax (cont.)

- Where does Python look for module source files?
  - Python is shipped with many modules ("batteries included") and they are all part of the Python installation
  - Modules that go with your main program should be in the same folder as the main program itself
  - If you have modules that is shared among multiple programs, you can either
    - install it in the Python installation location, or
    - set up your own module folder and modify **sys.path** or **PYTHONPATH**

Importing a Module

- Executing “import wc” the first time:
  - Creates a new module object
  - Executes the code in wc.py within the context of the new module
  - In the importing module, creates a variable named “wc”, which references the module object, for accessing the contents of the module
- Executing “import wc” again only does the very last step, ie the code in wc.py is **not** executed more than once

Module Context

- Python has the concept of contexts or namespaces for modules
  - Each module keeps track of its own set of variable names, so the same variable name in different modules refer to different variables
  - For example, each module has a variable named "__name__" which contains the name of the module
    - For the main program it has value "__main__"
    - For our wc module, it has value "wc"
  - The "def linecount(…)" statement in wc.py creates a function named "linecount" in the "wc" module

Module Context (cont.)

- To access a function or variable in another module, we need to specify both the module and function/variable name, eg **wc.linecount**
  - The **wc** part is the name of a *variable*, not the *module*!
  - We can do things like: "import wc as zzz" or "zzz = wc" and refer to **zzz.linecount**, but the *module* name is still **wc** (as witnessed by **zzz.__name__**)

Module Context (cont.)

- There are other forms of the import statement
  - **import** *module* **as** *myname*
    - This does the same thing as "import module" except the variable created in the importing module is named "myname" instead of "module"
  - **from** *module* **import** *name*
    - This creates a variable "name" in the importing module that refers to the same object as *module.name* **at the time when the import statement is executed**
    - This is mainly used to avoid having the imported module name appear many times in the code (either to reduce typing or to improve code readability)
    - You should only use this form with constants and functions, ie items that do not change value over time

Module Context (cont.)

- **from** *module* **import ***
  - For every variable or function in the imported module (whose name does not begin with _), a corresponding variable of the same name is created in the importing module
  - This was done frequently in the early days of Python to minimize typing
  - It is generally accepted that this is a **bad thing** to be avoided when possible because it destroys the name-clash protection of multiple namespaces and makes it difficult to track down where variables come from

Module Context (cont.)

- When a function executes, it looks for variable using the LSGB rule
  - L(ocal) variables defined in the function
  - S(cope) variables defined in enclosing functions
  - G(lobal) variables defined in the module
  - B(uilt-in) variables defined by Python
- The global variables refer to variables in the module where the function is **defined**, not the module where the function is **called**

Module Context (cont.)

- Example of functions and global variables

```
# Contents of gmod.py
var = 10

def print_var():
    print var

print_var()
```

```
# Using gmod.print_var
>>> var = 20
>>> import gmod
10
>>> gmod.print_var()
10
>>> from gmod import print_var
>>> print_var()
10
```

Using Modules

- Why use modules?
- Module is an organizational tool
  - Put related functions together into the same file
  - Avoid having multiple copies of the same code
- Functional decomposition
  - Put all code related to one task into a single file
  - markov2_prep.py, markov2_use.py
  - Main drawback is code duplication, eg **shift**
  - What if other programs also read the data files? Do we replicate **read_grams** in all of them?

Using Modules (cont.)

- How do we avoid duplicating code?
  - Put common code into files shared by multiple programs
- Modular programming
  - Put all code related to a subtask into a single file
  - markov3_io.py, markov3_prep.py, markov3_use.py
  - How do you choose the extent of a subtask?

Using Modules (cont.)

- On the plus side:
  - There is only one copy of the “shift” function
  - we no longer need to change either markov3_prep.py or markov3_use.py if we decide to use a different storage format; we just change markov3_io.py
- But… we still have to change all the files if we decide to use a different data structure for the prefix-suffix mapping, eg use a histogram instead of an expanded list of words
- Can we apply the shared module concept further to minimize work when changing code?

Using Modules (cont.)

- In markov3_use.py:
  - next_word = random.choice(m2[prefix])
- How do we interpret this statement?
  - Literally: choose a random value from the list of values that appear for key **prefix** in dictionary **m2**
  - Semantically: choose a random value from the list of words for that follow the two-word **prefix** using bigram-suffix mapping **m2**

Using Modules (cont.)

- We can use the statement:

```
next_word = random_suffix(m2, prefix)
```

  - instead of:

```
next_word = random.choice(m2[prefix])
```

- Assuming we:

```
def random_suffix(m, prefix):
  return random.choice(m[prefix])
```

- Why bother?
  - The reader gets a clearer idea of what is happening ("Oh, we're retrieving a random word following prefix.")
  - We can change how **random_suffix** is implemented (eg bias the word choice by the length of the word) without changing any other code in the program

Using Modules (cont.)

- Object-oriented programming (step 1)
  - Select a concept that can be represented as a collection of data structures
  - Group it together with the operations (functions) associated with the concept
  - Put the data structures and operations together and call the combination a "class" for the concept

Using Modules (cont.)

- Our markov3_*.py example has three files
  - markov3_prep.py reads a text file and generates two mappings: unigram-to-suffix and bigram-to-suffix
  - markov3_use.py uses the precomputed mappings to generate a partial sentence
  - markov3_io.py reads and writes the mappings
- What is a concept (and therefore candidate class) that spans the three files?

Using Modules (cont.)

- Concept: prefix-suffix mapping
  - We could have chosen to use two concepts: unigram-suffix mapping and bigram-suffix mapping
- We extract all data structures and operations on prefix-suffix mapping and put them into markov4_gram.py
- markov4_prep.py and markov_use.py are the same as their markov3 counterparts, but rewritten to use functions from markov4_gram.py (instead of accessing dictionaries directly)

Using Modules (cont.)

- Once the *prep* and *use* programs no longer directly access the mapping data, we are free to change how we represent the mapping data
- This is the separation of ***interface*** from ***implementation*** (aka **data abstraction** or **data encapsulation**)
  - Interface (aka API or application programming interface) is what callers of a module uses, eg functions and variables
  - Implementation is all the code within the module that makes using the interface work, eg code to update interface variables, and function definitions
  - ***As long as the module interface remains the same, the implementation may be changed at will***

Using Modules (cont.)

- Another way to look at it:
  - An API or interface defines what can be done semantically with a concept
  - An implementation is the underlying code that makes the semantic operations possible
  - A calling function should only care about the semantics and never about the underlying code
  - The underlying code may be changed as long as it re-implements the same or a superset of the API
    - Adding new functionality is fine
    - Removing or changing functionality is not

Using Modules (cont.)

- In our example, markov4_gram.py uses a redundant word list to represent possible suffixes for a given prefix
- We can change the implementation to using a word histogram and save a lot of memory
- In the new set of programs, notice that only markov5_gram.py differs from markov4_gram.py; markov5_prep.py and markov5_use.py are essentially identical to their markov4 counterparts

Class Syntax and Using Classes

- Note that in our example, we used only functions and modules to do object-oriented programming (OOP)
- Python (and many other languages such as C++ and Java) supports OOP by providing some extra constructs that aid bookkeeping
  - For example, each of our mapping is implemented using a single dictionary; there is no code to guarantee that we do not mistakenly use a unigram as the prefix for the bigram mapping
  - We can implement each mapping as a 2-tuple, with element 0 being the prefix length and element 1 being the dictionary, but this makes the code harder to read

Class Syntax

- Python provides a "class" syntax that allows us to group data together and access them by name

```
class class_name(object):
        """Documentation string"""
Instance1 = class_name()
instance1.first_attribute = first_value
print instance1.first_attribute
Instance2 = class_name()
instance2.second_attribute = second_value
print instance2.second_attribute
```

- The "(object)" part is not needed for Python 3

Class Syntax

- We can switch from dictionary to class syntax very easily
  - markov6_gram.py, markov6_prep.py, markov6_use.py

Class Syntax

- Classes are much more than just bookkeeping
- Next two weeks, more on classes and OOP
  - attributes and methods
  - initialization (constructor) and termination (destructor)
  - inheritance and polymorphism

Steps in Programming

- Figure out what problem you are solving
- Analyze the problem to identify concepts (divide and conquer)
- Figure out what data and functions are needed
- Write simplest code that solves the problem
- Write test code and debug
- Measure performance
- Optimize
  - Speed up hotspots
  - Change algorithms

Homework

- Assignment 6.1 - rectangles
  - Copy some code that use classes
  - Write some code that implement additional operations
- Assignment 6.2 – more rectangles
  - Write some code that calls the rectangle code
- What would you change to make the code more object-oriented?