Lecture Five

Putting it together

Plan for the day

- Quick review
  - `If/else` clauses
  - String replacement

- Writing usable code
  - Scripts
  - Modules
  - Functions

Control Flow: Selection

```
for item in collection:
     if condition:
          do something
```

Control Flow: Selection

```
for item in collection:
     if condition:
          do something
```

```
for item in collection:
     if condition:
          do something
     else:
          do something else
```

Control Flow: Selection

```
>>> L = [1,2,3,4]
>>> for i in L:
...  if i%2 == 0: # no remainder
...          print i
...
2
4
```

Control Flow: Selection

```
>>> for i in L:
...  if i%2 == 0:
...          print "%s is even!" % i
...  else:
...          print "%s is odd!" % i
...
1 is odd!
2 is even!
3 is odd!
4 is even!
```

Control Flow: Selection

- Multiple conditions

```
for item in collection:
    if condition:
        do something
    elif: # Read: "Else if"
        do something else
    else:
        do something when both are False
```

String replacement

- How do I print a variable within a string?

```
>>> for i in range(5):
...  print "Hello Dave, I'm on number %i" % i
...
Hello Dave, I'm on number 0
Hello Dave, I'm on number 1
Hello Dave, I'm on number 2
Hello Dave, I'm on number 3
Hello Dave, I'm on number 4
```

String replacement

- How do I print a variable within a string?

Place holder

```
>>> for i in range(5):
...  print "Hello Dave, I'm on number %i" % i
...
Hello Dave, I'm on number 0
Hello Dave, I'm on number 1
Hello Dave, I'm on number 2
Hello Dave, I'm on number 3
Hello Dave, I'm on number 4
```

String replacement

- How do I print a variable within a string?

Place holder

```
>>> for i in range(5):
...  print "Hello Dave, I'm on number %i" % i
...
Hello Dave, I'm on number 0
Hello Dave, I'm on number 1
Hello Dave, I'm on number 2
Hello Dave, I'm on number 3
Hello Dave, I'm on number 4
```

String replacement

- Need to specify which type you are replacing with
  - %i → int
  - %s → string
  - %d → numeric (captures ints and floats)

String replacement

- Multiple values

```
>>> D = {1: 'one', 2: 'two'}
>>> for i,j in D.iteritems():
...  print "Key: %d, Value: %s" % (i,j)
...
Key: 1, Value: one
Key: 2, Value: two
```

String replacement

- Multiple values

```
>>> D = {1: 'one', 2: 'two'}
>>> for i,j in D.iteritems():
...  print "Key: %d, Value: %s" % (i,j)
...
Key: 1, Value: one
Key: 2, Value: two
```

String replacement

- Multiple values

```
>>> D = {1: 'one', 2: 'two'}
>>> for i,j in D.iteritems():
...  print "Key: %d, Value: %s" % (i,j)
...
Key: 1, Value: one
Key: 2, Value: two
```

Types of programs

- Not everything happens at the interpreter
- It's good to have a copy of the code you ran
- There are at least three ways to interact with a written program
  - At the interpreter via `import`
  - Execution at the command line (scripting)
  - Execution with user input (`raw_input()`)

Scripts

- Series of commands in a text file
- Executed at the command line
- Will usually have text input and output

Scripts

```
Site,Observations,Species,Expenditure
Lake_Creek,4,12,180
Los_Alamos,8,340
Big_Bend,a,6,280
McDonald,5,20,280
Balmorrhea,3,3,174
```

Scripts

```
>>> line_list = []
>>> with open("homework.csv") as f:
>>>    for line in f:
>>>       line = line.strip().split(',')
>>>       line_list.append(line)

>>> expenses = {}
>>> for line in line_list[1:]:
>>>     expenses[line[0]] = line[-1]

>>> for site in expenses:
>>>    print "Spent %s at %s" % (expenses[site],site)
```

Scripts

```
>>> for site in expenses:
>>>    print "Spent %s at %s" % (expenses[site],site)
Spent 280 at Big_Bend
Spent 280 at McDonald
Spent 174 at Balmorrhea
Spent 340 at Los_Alamos
Spent 180 at Lake_Creek
```

Scripts

```
line_list = [] # Declare empty list
with open("homework.csv") as f: # Open file buffer
    for line in f:
        line = line.strip().split(',')
        line_list.append(line)

expenses = {}
for line in line_list[1:]:
    expenses[line[0]] = line[-1]

for site in expenses:
    print "Spent %s at %s" % (expenses[site],site)
```

Scripts

```
line_list = []
with open("homework.csv") as f:
    for line in f:
       line = line.strip().split(',')# Clean lines
       line_list.append(line)        # Build list

expenses = {}
for line in line_list[1:]:
    expenses[line[0]] = line[-1]

for site in expenses:
    print "Spent %s at %s" % (expenses[site],site)
```

Scripts

```
line_list = []
with open("homework.csv") as f:
    for line in f:
        line = line.strip().split(',')
        line_list.append(line)

expenses = {}    # Declare empty dictionary
for line in line_list[1:]:
    expenses[line[0]] = line[-1] # Populate from list

for site in expenses:
    print "Spent %s at %s" % (expenses[site],site)
```

Scripts

```
line_list = []
with open("homework.csv") as f:
    for line in f:
        line = line.strip().split(',')
        line_list.append(line)

expenses = {}
for line in line_list[1:]:
    expenses[line[0]] = line[-1]

for site in expenses: # Iterate over dictionary keys
    print "Spent %s at %s" % (expenses[site],site)
   # Use string replacement to print out a nice report
```

Scripts

- Now we want this functionality without having to retype at the interpreter.

- We're going to write it into a script
  a. Get rid of useless code
  b. Add 2 elements to make the script work
     - Telling the computer how to translate python
     - Provide input to the script
  c. Give ourselves permission to run the script (because we deserve it, and it's our computer, after all).
  d. Reap substantial rewards.

Scripts

```
line_list = [] # Turn this all into a list comprehension
with open("homework.csv") as f:
   for line in f:
       line = line.strip().split(',')
       line_list.append(line)

expenses = {}
for line in line_list[1:]:
    expenses[line[0]] = line[-1]

for site in expenses:
   print "Spent %s at %s" % (expenses[site],site)
```

Scripts

```
with open("homework.csv") as f: # You saved 3 lines
   line_list = [line.strip().split(',') for line in f]

expenses = {}
for line in line_list[1:]:
    expenses[line[0]] = line[-1]

for site in expenses:
   print "Spent %s at %s" % (expenses[site],site)
```

Scripts

```
with open("homework.csv") as f:
   line_list = [line.strip().split(',') for line in f]

expenses = {} # This part is useless, I just wanted to
for line in line_list[1:]: # make sure you understood
    expenses[line[0]] = line[-1] # dictionaries.

for site in expenses:
   print "Spent %s at %s" % (expenses[site],site)
```

Scripts

```
with open("homework.csv") as f:
   line_list = [line.strip().split(',') for line in f]

for line in line_list:
   print "Spent %s at %s" % (line[-1],line[0])
```

Scripts

```
with open("homework.csv") as f:
   line_list = [line.strip().split(',') for line in f]

for line in line_list:
   print "Spent %s at %s" % (line[-1],line[0])
```

- Note that you do not technically need to create line_list
  - You could iterate over the file, printing as you go.

Scripts

```
line_counter = 0
with open("homework.csv") as f:
   for line in f:
      line = line.strip().split(“,”)
      if line_counter > 0: # skip first line
         print "Spent %s at %s" % (line[-1],line[0])
      line_counter +=1 # increment line_counter
```

- Just add line_counter to skip first line.
  - This works, but why might you not want to do it?

Scripts

```
line_counter = 0
with open("homework.csv") as f:
   for line in f:
       line = line.strip().split(“,”)
       if line_counter > 0: # skip first line
          print "Spent %s at %s" % (line[-1],line[0])
       line_counter +=1 # increment line_counter
```

- Just add line_counter to skip first line.
  - This works, but why might you not want to do it?
  - The file has more information than just expenses, what if you want those in the future?

Scripts

- Now let’s see what the code looks like in a script

Scripts

- Now let's see what the code looks like in a script

- But now your script can only open the one file
  - This is called *hard coding* - avoid this by adding functionality to pass arguments to your script
  - Recall Unix

```
ls -a
Command argument
```

```
$ expenditures_1.py homework.txt
```

```
$ expenditures_1.py homework.txt
```

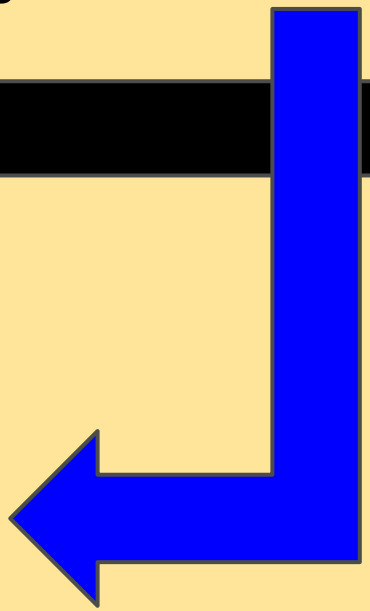
```
$ expenditures_1.py homework.txt
```

Script body

```
$ expenditures_1.py homework.txt
```

```
Script body
import sys
```

```
$ expenditures_1.py homework.txt
```



```
Script body
import sys

infile=sys.argv[1]
```

Modules

- A lot of people have developed widgets and extensions for use with Python
  - Some come with Python standard
  - Some you have to download

- How to get access from a script
  - `import`: bring module into your *name space*
  - Once there, access function via dot notation, as usual (yes, they are objects too).

```
import module
module.function()
```

Modules

sys	Interact with command line
os	More extensive interaction
pprint	“Pretty print”
itertools	Very useful combinatorics
math	Math
numpy, scipy, matplotlib, pandas	Linear algebra, stats, visualization, and much more - next week
BioPython	Excellent resources for bioinformatics

Modules

- Can I make my own modules?
  - We already have
  - Each python file is a potential module
  - Try this: `import expenditures_1`
  - Note: a file called expenditures_1.pyc will be created. This is a binary file, so the import will be faster next time.

- The next step is about how to do this better

In-class activity

- But first, write and execute a script called “species.py” that does what expenditures_2.py does, but for species.
  - Note: Los Alamos is missing one column, let’s say it’s missing “observations,” not “species.”

- Use your plain text editor, not Word.
  - Nano works, but will be a pain.

Functions

- When we imported expenditures_1.py, it just ran the script.
  - Afterwards, we could not access it’s functionality.

- We want another object that can hold functionality, without executing sequentially
  - These are functions

Functions

```
with open(infile) as f: # open and parse file
    line_list = [line.strip().split(',') for line in f]

for line in line_list[1:]: # print expenditures
    print "Spent %s at %s" % (line[-1],line[0])
```

Functions

```
def parse_file(infile): # open and parse file
    with open(infile) as f:
       line_list = [line.strip().split(',') for line in f]
    return line_list

def print_exps(infile): # print expenditures
    lines = parse_file(infile)
    for line in lines[1:]:
       print "Spent %s at %s" % (line[-1],line[0])
```

Functions

```
def parse_file(infile):
    with open(infile) as f:
       line_list = [line.strip().split(',') for line in f]
    return line_list

def print_exps(infile):
    lines = parse_file(infile)
    for line in lines[1:]:
       print "Spent %s at %s" % (line[-1],line[0])
```

- `Def`: Like a variable, binds indented code to a name
  - Prevents execution until it is called by name

Functions

```
def parse_file(infile):
    with open(infile) as f:
       line_list = [line.strip().split(',') for line in f]
    return line_list

def print_exps(infile):
    lines = parse_file(infile)
    for line in lines[1:]:
       print "Spent %s at %s" % (line[-1],line[0])
```

- `Return`: Output a value without printing it. The value can now be bound to variable.

Functions

```
def parse_file(infile):
    with open(infile) as f:
       line_list = [line.strip().split(',') for line in f]
    return line_list

def print_exps(infile):
    lines = parse_file(infile)# Call parse_file, and bind
    for line in lines[1:]:    # its output to name 'lines'
       print "Spent %s at %s" % (line[-1],line[0])
```

- `Return`: Output a value without printing it. The value can now be bound to variable.

Functions

```
## expenditures_3.py - let's see how it executes
import sys
infile = sys.argv[1]

def parse_file(infile):
    with open(infile) as f:
        line_list = [line.strip().split(',') for line in f]
    return line_list

def print_exps(infile):
    lines = parse_file(infile)
    for line in lines[1:]:
        print "Spent %s at %s" % (line[-1],line[0])

print_exps(infile)
```

Functions

```
import sys # Import
infile = sys.argv[1]

def parse_file(infile):
    with open(infile) as f:
       line_list = [line.strip().split(',') for line in f]
    return line_list

def print_exps(infile):
    lines = parse_file(infile)
    for line in lines[1:]:
       print "Spent %s at %s" % (line[-1],line[0])

print_exps(infile)
```

Functions

```
import sys
infile = sys.argv[1] # Declare global variable 'infile'

def parse_file(infile):
    with open(infile) as f:
       line_list = [line.strip().split(',') for line in f]
    return line_list

def print_exps(infile):
    lines = parse_file(infile)
    for line in lines[1:]:
       print "Spent %s at %s" % (line[-1],line[0])

print_exps(infile)
```

Functions

```
import sys
infile = sys.argv[1]

def parse_file(infile): # Declare parse_file
    with open(infile) as f:
       line_list = [line.strip().split(',') for line in f]
    return line_list

def print_exps(infile):
    lines = parse_file(infile)
    for line in lines[1:]:
       print "Spent %s at %s" % (line[-1],line[0])

print_exps(infile)
```

Functions

```
import sys
infile = sys.argv[1]

def parse_file(infile):
    with open(infile) as f:
       line_list = [line.strip().split(',') for line in f]
    return line_list

def print_exps(infile): # Declare print_exps
    lines = parse_file(infile)
    for line in lines[1:]:
       print "Spent %s at %s" % (line[-1],line[0])

print_exps(infile)
```

Functions

```
import sys
infile = sys.argv[1]

def parse_file(infile):
    with open(infile) as f:
       line_list = [line.strip().split(',') for line in f]
    return line_list

def print_exps(infile):
    lines = parse_file(infile)
    for line in lines[1:]:
       print "Spent %s at %s" % (line[-1],line[0])

print_exps(infile)  # Call to print_exps
```

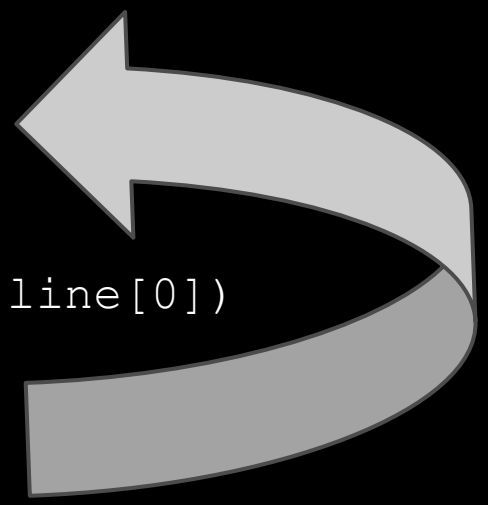
Functions

```
import sys
infile = sys.argv[1]

def parse_file(infile):
    with open(infile) as f:
       line_list = [line.strip().split(',') for line in f]
    return line_list

def print_exps(infile): # Execute
    lines = parse_file(infile)
    for line in lines[1:]:
       print "Spent %s at %s" % (line[-1],line[0])

print_exps(infile)  # Call to print_exps
```



Functions

```
import sys
infile = sys.argv[1]

def parse_file(infile): # Execute
    with open(infile) as f:
       line_list = [line.strip().split(',') for line in f]
    return line_list

def print_exps(infile):
    lines = parse_file(infile)# Call
    for line in lines[1:]:
       print "Spent %s at %s" % (line[-1],line[0])

print_exps(infile)
```

Functions

```
import sys
infile = sys.argv[1]

def parse_file(infile):
    with open(infile) as f:
       line_list = [line.strip().split(',') for line in f]
    return line_list # Return

def print_exps(infile):
    lines = parse_file(infile)# Bind
    for line in lines[1:]:
       print "Spent %s at %s" % (line[-1],line[0])

print_exps(infile)
```

Functions

```
import sys
infile = sys.argv[1]

def parse_file(infile):
    with open(infile) as f:
       line_list = [line.strip().split(',') for line in f]
    return line_list

def print_exps(infile):
    lines = parse_file(infile)
    for line in lines[1:]: # Print
       print "Spent %s at %s" % (line[-1],line[0])

print_exps(infile)
```

Functions

- We still have a problem with `import`

```
>>> import expenditures_3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "expenditures_3.py", line 5, in <module>
    infile = sys.argv[1]
IndexError: list index out of range
```

- Choked on global variable, because we had no input from command line.

Functions

- Solve this by putting this at the bottom of the script (expenditures_4.py):

```
if __name__ == '__main__':
    infile = sys.argv[1]
    print_exps(infile)
```

- "If I (__name__) am being executed from the command line (__main__), do the below"
  - Protects the functions from execution unless the script is being executed at the command line.

Functions

- Finally!

```
>>> import expenditures_4 # Interpreter
>>> expenditures_4.print_exps("homework.csv")#Dot notation
Spent 180 at Lake_Creek
Spent 340 at Los_Alamos
Spent 280 at Big_Bend
Spent 280 at McDonald
Spent 174 at Balmorrhea

$ ./expenditures_4.py homework.csv # Command line
Spent 180 at Lake_Creek
Spent 340 at Los_Alamos
...
```

Functions

- Use file redirection

```
$ ./expenditures_4.py homework.csv >> expense_summary.txt
```

- Or you could open a file to print to within the script.
  - But make sure to avoid hard coding, i.e., use the second command line argument (`sys.argv[2]`) to name the outfile

Functions

- We made this code more streamlined, modular, and readable.

- It can be used as a module or a script
  - In module form, it’s script name becomes its module name (minus the “.py”) and its functions become module methods.

More on functions

- Functions should do one, modular task
  - Think about what you want to do first

More on functions

- Functions should do one, modular task
  - Think about what you want to do first

- Write out the steps you think your code should follow:
  - "Open and parse file into a list"
  - "Loop over list and extract x, y, but not z"

    etc etc...

More on functions

- Then open a file, put in a SheBang line and…

- Start writing functions!
  - Start at the bottom
  - “Open and parse file into a list” becomes…

```
def parse(infile):
   with open(infile) as f:
      return parsed data
```

More on functions

- Then open a file, put in a SheBang line and...

- Start writing functions!
  - Start at the bottom
  - Now you need a new function: `parsed_data()`

```
def parse(infile):
   with open(infile) as f:
      return parsed_data(infile)
```

More on functions

- Then open a file, put in a SheBang line and...

- Start writing functions!
  - Start at the bottom

```
def parsed_data(buffer):
   parsed_object = object
   for line in buffer:
      parsed = line_parser(line) # etc. etc.
      add parsed to parsed_object
```

More on functions

- Functions should be direct expressions of the flow you want your code to take
  - Organizing them is the hard part
  - Don’t expect to get it right the first time

- Writing it all out by hand first helps
  - Better yet, when you’re writing, also write down a test that each step should pass

```
parsed(infile)
#should return a list of lists
```

More on functions

- Passing more than one argument

```
def function(parameter list):
   code to be executed
```

- *parameter list* is a comma delimited series of objects you wish to *pass* to the function.
- Can set default values

```
def parsed(infile="homework.csv"):
   return parsed infile
```

Common pitfalls

- Variable scope

```
def parse_file(infile):
    with open(infile) as f:
        line_list = [line.strip().split(',') for line in f]
    return line_list

def print_exps(infile):
    for line in line_list: # Why can't I do this?
        print "Spent %s at %s" % (line[-1],line[0])
```

- `line_list` has local *scope* within `parse_file`

Common pitfalls

- Variable scope

```
def parse_file(infile):
    with open(infile) as f:
        line_list = [line.strip().split(',') for line in f]
    return line_list

def print_exps(infile):
    lines = parse_file(infile) # That's why we need to pass
    for line in lines[1:]:
        print "Spent %s at %s" % (line[-1],line[0])
```

Program Flow

- Ideally, programs are cascading sets of functions that are not hard-coded
  - When you're structuring a program, it's important to think about who will use the program. Why will they use it? How can you make the program more flexible?

raw_input()

- We talked about sys_argv[]
- What if you want to have someone input some value for a calculation
- Python has a function for this called raw_input()
- This will take in a value that can be interacted with by a script

raw_input()

- >>> a = raw_input('Please enter a number here: ')

  >>> print a

raw_input()

- >>> a = raw_input('Please enter a number here: ')

  >>> print a

Please enter a number here:

raw_input()

- >>> a = raw_input('Please enter a number here: ')

  >>> print a

Please enter a number here: 12

raw_input()

- >>> a = raw_input('Please enter a number here: ')
  >>> print a

Please enter a number here: 12

12

raw_input()

- So what happened here?
  - Python read the raw_input call and prompted you to enter some information
  - Python read this information and did what you said to do with it
    - Print, in this case
  - But you could do pretty much any other operation

raw_input()

- What if I had entered a letter?
  - raw_input would have accepted it
  - This is why it's helpful to have text that tells the user what to put in

Wrapping it up and putting a bow on it

- When do you want to write to a file versus to the standard output?

Wrapping it up and putting a bow on it

- When do you want to write to a file versus to the standard output?
  - Standard out is great for including print statements to do error checking
  - Also for passing output to other programs or scripts

Wrapping it up and putting a bow on it

- When do you want to write to a file versus to the standard output?
  - Standard out is great for including print statements to do error checking
  - Also for passing output to other programs or scripts
  - Writing to a file is great if you need to run part of your script in one location and part in another
    - Generate data file on desktop, Run on TACC
  - Temporal separation of steps.
  - Import to R.

Wrapping it up and putting a bow on it

- When do you want to write to a file versus to the standard output?
  - Standard out is great for including print statements to do error checking
  - Also for passing output to other programs or scripts
  - Writing to a file is great if you need to run part of your script in one location and part in another
    - Generate data file on desktop, Run on TACC
  - Temporal separation of steps.
  - Import to R.
  - Some of this is personal; I output nearly everything to file so I have a constant record of my activities

Homework

- Look up the modules we gave you above.
  - Google “python *module*”
  - See which ones look interesting
  - If you’re feeling plucky, try some out
  - Make note of anything you might like us to cover on the free day, and email us with this info (be specific)
- Read the Cheatsheet and Extensions
- Take the script homework.py and break it up into functions
  - It should be executable from the command line and importable at the interpreter.