COMS 3101-3 Programming Languages – Python: Lecture 2

Kangkook Jee

jikk@cs.columbia.edu

Logistics

- TA assigned
  - Yuhan Zhang (yz2637@columbia.edu)
    - CS MS student
  - Office hour: Mon 1:00 – 3:00 PM
- Reminders for newly registered students
  - Submit HW 1 with HW 2
  - Office hours: Tue, Fri 11:00 ~ 2:00 PM
  - Most class materials from course website
    - http://www.cs.columbia.edu/~jikk/teaching/3101-3

Project

- Important Dates
  - Proposal due in two weeks (Sep 26)
  - Final Project due (Oct 17)
- Project must be done in team of 2 ~ 3
- Project should be larger than the weekly homework problems, yet feasible in 3 weeks
- Something fun or something that's useful or interesting to you
- Proposal + deliverable = 50% of final grade

Sample Project Ideas

- A web crawler that harvests meaning information from the web and process it (news headline aggregator)
- A package that does something interesting using publicly open APIs from social networking services
  - e.g., Facebook, Flickr, Twitter
- Traffic analysis using Google maps API
- A package with API to analyze a specific data set
  - For whatever data you're interested in
  - Data mining Google Stock data
  - Data mining IMDB data
- … (Be creative!)

Project proposal

- Due in two weeks
  - 10% of final grade
- Should be about 2 pages
  - List team members
  - High level description of scope and purpose of the project
  - Draft of how project will be broken up into packages, modules, and important classes
  - Short description of planned work-flow in your team
    - How will the work be divided?
    - How will the components be combined?
    - How will you do testing?

Review

- Variable and object
  - Everything is object in Python
  - variables are names mapped to objects
  - value comparison (‘==‘) object comparison (‘is’)
- Control flows
  - conditional-statment: `if` statement (no switch statement)
  - loop-statement: while, for statement
  - break, continue
  - optional else statement
- Basic data types
  - Boolean type: `True, False`
  - number types : `int, long, float`
- Advanced types
  - `list(`mutable), `tuple`(immutable)
- Common sequence type operations
  - testing, slicing, iteration with `for`-loop statement

Agenda

- Advanced data types
  - More list
  - Dictionaries
  - Sets
  - Strings
- File I/O
- Functions

Quiz

- For given *n*, what the statement would produce?

```
[i  for i in range(n) for j in range(2, i)
if i % j == 0 ]
```

ADVANCED DATA TYPES: LIST, DICTIONARY, SET, STRING

range()

- `range(i)` produces the list of integers from 0 to `i` (exclusive).

```
>>> range (10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- `range(i,j)` produces the list of integers from `i` (inclusive) to `j` (exclusive).

```
>>>range(-3,4)
[-3, -2, -1, 0, 1, 2, 3]
```

- `range(i,j,s)` produces the list of integers from `i` (inclusive) to `j` (exclusive) in step of `s`.

```
>>> range (10,1,-2)
[10, 8, 6, 4, 2]
```

List Comprehension

- Perform some operation each element of an iterator and get a new list

```
[ expr1 for x in sequence if condition]
```

```
>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]
>>> [2 ** x for x in ran
[1, 2, 4, 8, 16, 32, 64,
```

- Can use multiple for state
  - Equivalent to double for-loop

```
l = []
for a in [1,2]: #range(1,3)
   for b in ['a', 'b']:
      l.append((a, b))
```

```
>>> # Compute all pairs
... [(a,b) for a in range(1,3) for b in
['a','b']]
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
```

else in List Comprehension

- Can use conditional expression within a list comprehension

```
[ expr1 for x in sequence if condition]
```

```
>>> [a if a % 2 == 0 else 'bleep’ \
... for a in range(10)]
[0, 'bleep', 2, 'bleep', 4, 'bleep', 6, 'bleep', 8,
'bleep']
```

- if-else code block is part of **`expr1`**
  - if does not filter the iteration in this case!

List Operations (1)

- List members can be accessed with array indexing

```
>>> a = ['fuji', 'gala']
>>> a[1]
'gala'
>>> a
['fuji', 'gala']
```

- Lists are mutable and can be manipulated
- `list.append(x)` adds element x to the end of list
  - Note: `list += [x]` will return a new list
- `list.insert(n, x)` inserts element x before *n*-th element

```
>>> a.append('macintosh')
>>> a
['fuji', 'gala', 'macintosh']
>>> a.insert(0, ‘honeycrisp’)
>>> a
['honeycrisp’, 'fuji', 'gala’, ‘macintosh]
```

List Operations (2)

- `list.pop()` removes the last element from list and returns it.
  - list.pop([index]) can take optional argument
  - Lists can be used as stacks: append() performs push operation

```
>>> a.pop()
'macintosh'
>>> a
['honeycrisp', 'fuji', 'gala']
```

- `list.remove(x)` removes the first occurrence of element x from the list.

```
>>> a.remove('fuji')
>>> a
['honeycrisp', 'gala’]
```

List Operations (3)

- `list.reverse()` reverses the order of the list.

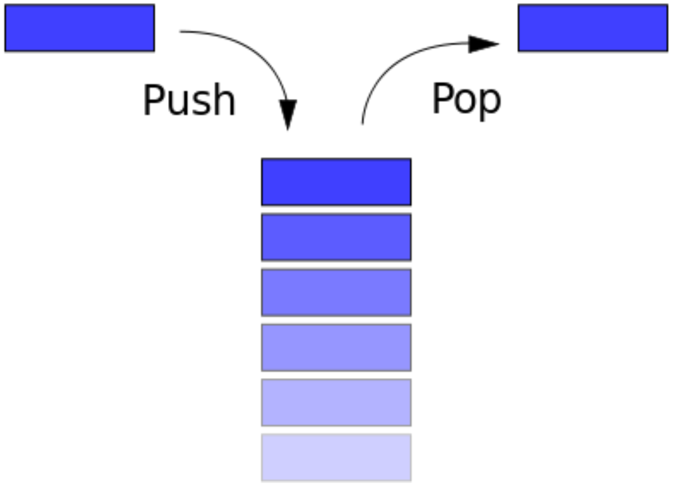
```
>>> a
['honeycrisp', 'fuji', 'gala', 'macintosh']
>>> a.reverse()
>>> a
['macintosh', 'gala', 'fuji', 'honeycrisp']
```

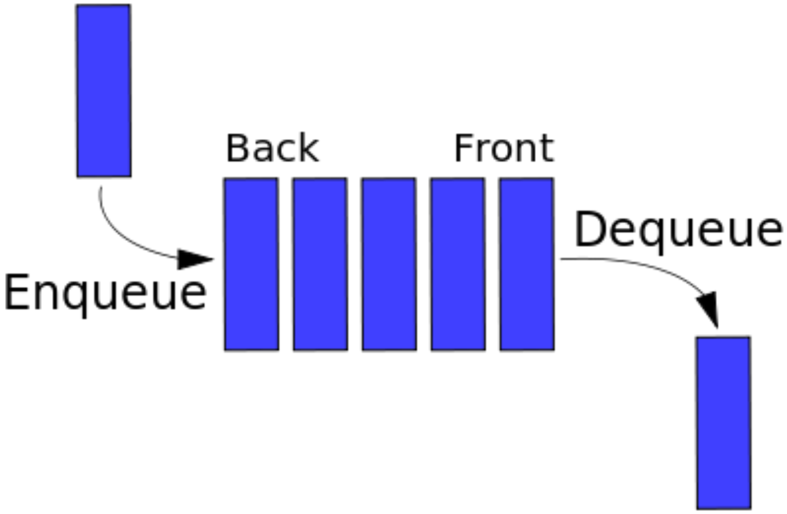
- `list.sort()` sorts the list (using <=)

```
>>> a.sort()
>>> a
['fuji', 'gala', 'honeycrisp', 'macintosh']
```

Stack and Queue

- Basic CS data type abstractions
- Stack
  - Last in first out (LIFO)
  - Operations: push(x), pop()
- Queue
  - First in first out (FIFO)
  - Operations: enqueue(x), dequeue()
- Both can be implemented with Python list





Dictionaries

- A dictionary is a collection of objects indexed by unique keys
  - Equivalent concepts： *hash*， *map* data structures

```
>>> legs = {'cat':4, 'human':4, 'centipede':100}
>>> legs['cat']
4
```

- Assigning a new object to an unseen key inserts the key into the dictionary

```
>>> legs['python'] = 0
>>> legs
{'python': 0, 'centipede': 100, 'human': 4, 'cat': 4}
```

- Keys are hashed (they must be ***immutable***)
  - e.g., tuple, str, numbers:⭕, list, dictionary, set: ❌
- Values can be *any* objects

Testing for Membership

- `x in dict` returns `True` if x is a key of dict, `False` otherwise

```
>>> legs
{'python': 0, 'centipede': 100, 'human': 4, 'cat': 4}
>>> 'cat' in legs
True
>>> 'lion' not in legs
True
```

- use `del` statement to remove a key, value pair

```
>>> del legs['python']
>>> legs
{'centipede': 100, 'human': 4, 'cat': 4}
```

Dictonary Items, Keys and Values

- `dict.keys()` get a list of keys

```
>>> legs = {'cat':4, 'human':4, 'centipede':100}
>>> legs.keys()
['centipede', 'human', 'cat']
```

- `dict.values()` get a list of values

```
>>> legs.values()
[0, 100, 4, 4]
```

- `dict.items()` get a list of (key, value) tuples

```
>>> items = legs.items()
>>> items
[('centipede', 100), ('human', 4), ('cat', 4)]
```

- `dict(x)` constructs dictionary with a list of item tuples

```
>>> dict(items)
{'centipede': 100, 'human': 4, 'cat': 4}
```

Sets

`set` (mutable) / `fronzenset` (immutable) are unordered bags of unique objects

```
>>> s = set(['lion','tiger','panther'])
```

- set membership: `x in s`

```
>>> 'tiger' in s
True
>>> 'Tiger' in s
False
```

- is a subset / superset?

```
>>> frozenset(['lion', 'panther', 'tiger']) <= s
True
>>> frozenset(['lion', 'panther', 'tiger']) < s
False
```

Sets: Union/Intersection/Difference

- Get union of `s` and `t` as a new set

```
>>> set(['a', 'b']) | set(['c', 'd'])
set(['a', 'c', 'b', 'd'])
```

- Get intersection of `s` and `t` as a new set

```
>>> set(['a', 'b']) & set(['a', 'c'])
set(['a'])
```

- Get difference between `s` and `t` as a new set

```
>>> set(['a', 'b']) - set(['a', 'c'])
set(['b'])
```

Sets - Mutable operations: update, add, remove

- Add all elements of a set to set s

```
>>> s = set(['a','b'])
>>> s.update(set(['a','c']))
>>> s
set(['a', 'c', 'b'])
```

- Add object x to s

```
>>> s.add('a')
>>> s.add('d')
>>> s
set(['a', 'c', 'b', 'd'])
```

- Remove object x from s

```
>>> s.remove('b')
>>> s
set(['a', 'c', 'd'])
```

Quiz: Solution

- For given *n,* what the statement would produce?

```
[i  for i in range(n) for j in range(2, i)
if i % j == 0 ]
```

```
>>> l = []
>>> for i in range(n):
...     for j in range(2, i):
...         if i % j == 0:
...             l.append(i)
```

- return a list of non-prime numbers below n
- The following returns a set of prime integers below n

```
set(range(3, n)) - set([i  for i in range(n) for
j in range(2, i)  if i % j == 0 ])
```

String Literals (1)

- String literals can be defined with a single quotes or double quotes
- Can use other type of quotes inside the string

```
>>> s1 = 'hello "COMS3101-1”'
>>> print(s1)
hello "COMS3101-1”
```

- Can use ''' or """ to delimit multi-line strings

```
>>> s = '''
... Hello
... "COMS3101-1"
... '''
>>> print(s)
Hello
"COMS3101-1"
```

String Literals (2)

- Some special characters need to be ***escaped***

```
#single quotes inside single quote
>>> print ('Hello \'COMS3101-1\'')
Hello 'COMS3101-1’
>>> print ('Hello \\ COMS3101-1')  # Backslash
Hello \ COMS3101-1
>>> print ('Hello \n COMS3101-1')  # Newline
Hello
 COMS3101-1
>>> print ('Hello \t COMS3101-1')  # Tab
Hello     COMS3101-1
```

String Operations - Review

- Strings support all sequence operations

```
>>> len('foo')  # Get length
3
>>> 'a' * 10 + 'rgh'  # Concatenation and repetition
'aaaaaaaaaargh'
>>> 'tuna' in 'fortunate'  # Substring
True
>>> 'banana'.count('an')  # Count substrings
2
>>> 'banana'[0]  # index operation
'b’
>>> 'banana'.index('na')  # Find index
2
>>> 'banana'[2:-1]  # slicing
'nan'
```

- Also support iteration and list comprehension

Additional String Operations (1)

- Capitalize first letter, convert to upper/lower case or Tile Case.

```
>>> 'grail'.capitalize()
'Grail'
>>> 'grail'.upper()
'GRAIL'
>>> 'GRAIL'.lower()
'grail’
>>> 'the holy grail'.title()
'The Holy Grail'
```

- Check whether the sting *starts* or *ends* with a string

```
>>> "python".startswith("py")
True
>>> "python".endswith("ython")
True
```

Additional String Operations (2)

- Split a string into a list of its components using a separator

```
>>>#Separate on whitespace, tabs, linefeeds
... "An African\t or European\n swallows?".split()
['An', 'African', 'or', 'European', 'swallows?’]
>>>#Can specify custom separator string
... "python, java, lisp, haskell".split(",")
['python', ' java', ' lisp', ' haskell’]
```

- Join together a sequence of strings using a separator string

```
>>>#Format a list in CSV format:
...','.join(['join', 'a', 'list', 'of', 'string'])
'join,a,list,of,string'
```

Additional String Operations (3)

Simple tests on strings

- Contains only digits?

```
>>> '42'.isdigit()
True
```

- Contains only upper/lowercase letters?

```
>>> 'Alphabet'.isalpha()
True
```

- Contains only digits and upper/lowercase letters?

```
>>> ’253engineering'.isalnum()
True
```

Use regular expressions (‘re’ module) for advanced pattern matching

String Formatting (1)

- For pretty-print data or to write it to a file

```
formatstr.format(argument_0, argument_1, ...)
```

replaces place holders in `formatstr` with arguments

- Place holder `{i}` is replaced with the `i`-th argument
  - Arguments are *implicitly* converted to str.

```
>>> s = "{0}, {1}C, Humidity: {2}%”
>>> s.format('New York', 10.0, 48)
'New York, 10.0C, Humidity: 48%’
>>> s = "{temp}C"  # can assign names to format fields
>>> s.format('New York', 48, temp=10.0)
'10.0C’
>>> # Literal { need to be escape by duplication
... "{{ {temp}C }}”.format('New York', 48, temp=10.0)
'{ 10.0C }’
```

String Formatting (2)

- If an argument is a sequence, can use index operation in format string

```
>>> "{0[0]}, {0[1]}, and {0[2]}".format(['1st','2nd','3rd'])
'1st, 2nd, and 3rd'
```

- Place holders can contain format specifiers (after a :)
  - specify minimum field width and set alignment
  - number formatting for precision, exponentation, percentage

```
>>> "|{0:^5}|{1:<5}|{2:>5}|".format("x","y","z")
'|  x  |y    |    z|’
>>># Percentage with single decimal
..."{0:.1%}".format(0.1015)
'10.2%'
```

FILE I/O

File Objects

- To read or write a file has to be opened
- open (filename, [more]) returns object of type file
- Allows read, write, append operations
  - ‘mode: r’: read only, ‘w’: write only, ‘a’: append at the end
  - appending ‘b’ to the mode opens file in binary mode
- After operations, file object need be closed

```
>>> f = open('/etc/passwd', 'r')
>>> f
<open file '/etc/passwd', mode 'r' at 0x110204930>
>>> f.close()
```

Reading from Text Files – Line Reading

File nee.txt:

```
ARTHUR: Who are you?
KNIGHT: We are the Knights Who Say... Nee!
```

- Return a single line every time `file.readline()` is called (including \n)
- `readline()` returns an empty string if there is no more line to read

```
>>> f = open('nee.txt', 'r')
>>> l = f.readline()
>>> while l:
...     print(l)
...     l = f.readline()
...
ARTHUR: Who are you?

KNIGHT: We are the Knights Who Say... Nee!
å
```

print() add linebreaks (new line character)

Reading from Text Files – Multiple lines

File nee.txt:

```
ARTHUR: Who are you?
KNIGHT: We are the Knights Who Say... Nee!
```

- `f.readlines()` returns a list of all lines

```
>>> f = open('nee.txt', 'r')
>>> lines = f.readlines()
>>> f.close()
>>> for l in lines:
...     print(l)
...
ARTHUR: Who are you?

KNIGHT: We are the Knights Who Say... Nee!
```

Reading from Files – read() and seek()

File test.txt:

```
This is a test.
```

- `f.read([size])` reads (at most) the next size bytes
  - if size is not specified, the whole file is read
  - returns empty string if no more bytes available
- f.seek(offset) jumps to position offset in the file

```
>>> f = open('test.txt', 'r')
>>> f.read()
'This is a test.\n'
>>> f.seek(0)  # reset file pointer to the beginning
>>> s = f.read(10)
>>> while s:
...     print s
...     s = f.read(10)
...
This is a
test.
```

Writing to Files

- `f.write(str)` writes str to the file
- `f.writelines(iter)` writes each string from an iterator to a
  `f.close()` commits everything to file from buffer
- `f.flush()` to force commit without closing

```
>>> f = open('test2.txt', 'w')
>>> f.write('hello!')
>>> f.writelines(['a','b','c'])
>>> f.close()
```

File test2.txt:

```
hello!abc
```

FUNCTIONS

Functions

- Programing abstraction that computes and returns some result, given its parameters

```
def pythagoras(leg_a,leg_b):
    """ Compute the length of the hypotenuse
    opposite of the right angle between leg_a
    and leg_b.
    """
    hypotenuse = math.sqrt(leg_a**2 + leg_b**2)
    return hypotenuse
```

```
>>> pythagoras(3,4)  # function call with arguments
5.0
```

  - Concise and clear code: breaks up code into meaningful units by avoiding duplicate code
  - Abstract away from concrete problem
  - Can be shared/re-used through modules
  - Powerful computation device: allow recursion

Function Definitions

```
def function_name(param_1, ..., param_n):
    “””
    A docstring describing the function.
    “””
    statements
    return result
```

- Conventions for function name and parameters: lower_case_with_underscore
- Docstring parameters, and return are optional
- return can occur anywhere in the function
  - Terminates the function and returns the return value (or None if no value is provides)
  - A function with no return statement returns None once if there are no more statements to execute

Function Calls

```
>>> def append(x, lst):
...     lst.append(x)
>>> append
<function append at 0x104a18de8>
>>> l = [‘a’]
>>> append(‘b', l)
>>> print(l)
['a', ‘b']
```

- When a function is called, arguments are passed through its formal parameters
- The parameter names are used as variables inside the function
- Call by object: parameters are names for objects

Parameters with Default Value

```
>>> def append(x, lst=[]):
...     lst.append(x)
...     return lst
...
>>> append('a')
['a']
>>> # Watch out for mutable objects in default parameters
...append('b')
['a', 'b']
```

- Function definiation can assign default values to parameters
- When no argument is passed during a function call, default value is assumed
- Default values are computed when function is defined!
  - Not from the call site
-

Extra Positional and Named Arguments

```
>>> def foo(*args, **kwargs):
...     print type(args)
...     print type(kwargs)
...     print args
...     print kwargs
...
>>> foo(1, 2, 3, a=1, b=2, c=3)
<type 'tuple'>
<type 'dict'>
(1, 2, 3)
{'a': 1, 'c': 3, 'b': 2}
```

- Two ways to support arbtrary length input parameters
  - Poisional arguments: `*args(tuple)` defines a tuple of additional positional arguments
  - Named arguments: `**kwargs (dictionary):` defines an dictionary of additional named arguments with parameter name and value pairs

Scope

- Scope: the part of program where variable's definition is visible and valid
- Variables visable from function inside
  - Function parameters, locally defined variables
  - Variables defined parent scope (read-only)
- Re-assigning them creates a new local variable
  - Changes to the variable is not visible to outside

```
a=1
def foo(b):
    c=2      # local c
             # visible variables: global a, parameter b, local c

def bar(b):  # different b
    c = 3    # different c
             # global a is visible thus far
    a = 3    # create new local variable a
             # visible variables: parameter b, local c, local a

# global a visible again, cannot see either b or c
```

Scope: what does program prints? (1)

```
x = 3

def foo():
    print(x)

x = 2

def spam(x):
    print(x)

def bar():
    x = 7
    print(x)

def eggs():
    print(x)
    x = 5
```

```
print(x)        2

foo()           2

spam(9)         9

print(x)        2

bar()           7

eggs()
```

UnboundLocalError: local variable 'x' referenced before assignment

Scope: what does program prints? (2)

```
x = 3

print(x)                                     3

for x in range(2):  # [0,1]                  0
    print (x)                                1

print (x)                                    1
```

- Block structure (specially loops) does not define scope!

Nested Functions

```
>>> def a():
...     print('spam')
…
>>> def b():
...     def a():
...         print('egg')
...     a()
...
>>> a()
spam
>>> b()
egg
>>> a()
spam
```

- Function definitions can be nested
- Function names are just variables bound to function object (first-class functions).
- Therefore the same scoping rules as for variables apply

Function Closures

- Nested functions can be used to create closures
- Closure: Function object that contains some ‘state’
  - Referring to variables from outside function’s local scope when function object is created

```
>>> def make_power_func(x):
...     def power(y):
...         return y**x
...     return power
...
>>> power_two = make_power_func(2)
>>> power_two(16)
256
```

x is in the surrounding scope of function power. Its binding is preserved since function power is defined (i.e., when make_power_func is called with concrete value x)

Backup slides

stdin and stdout

- Can access terminal input (sys.stdin) and terminal output (sys.stdout) as file object
- These objects are defined globally in the module ‘sys’
  - ‘sys’ is loaded with import statement

```
>>> import sys
>>> sys.stdout.write('Hello world!\n')
Hello world!
>>> sys.stdin.read(4)
COMS3101-3
'COMS'
```

File Operation with ‘os’

- ‘os’ module defines interfaces that enable interactions with operating systems
  - Most frequently used component of standard library
  - Implements majority subset of OS system call API

```
>>> import os
>>> os.system('date')  # OS specific command
Wed Sep 9 22:16:59 EDT 2013
0
```

- ‘os.path’ sub-module defines interfaces for filename manipulation

```
>>> os.path.isdir("/tmp")  # some folder
True
```

os.path Module – manipulate pathnames

- os.path.abspath(path) – Returns the absolute pathname for a relative path

```
>>> os.path.abspath('python')
'/opt/local/bin/python'
```

- os.path.basename(path) – Returns the absolute pathname for a relative path

```
>>> os.path.abspath('python')
'/opt/local/bin/python'
```

- os.path.getsize(path) – Returns the size of path in byte

```
>>> os.path.getsize("python")
13404
```

- os.path.isfile(path) – Returns True if the path points to a file
- os.path.isdir(path) – Returns True if the path points to a directory

os Module – list content of a directory

- For homework: os.listdir(path) lists files in a directory

```
>>> os.listdir("/tmp")
['.font-unix', '.ICE-unix', ... , android-jikk']
```

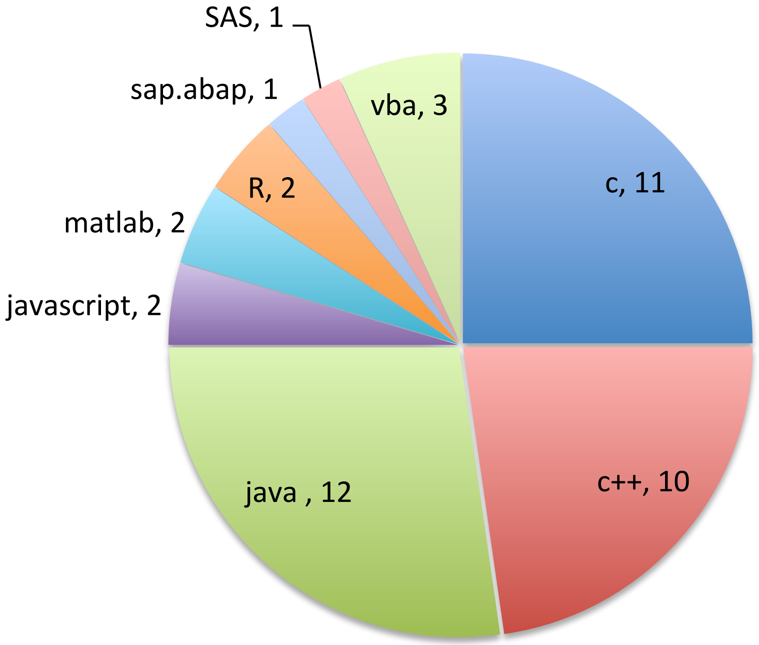
Class Statistics

By Major

Major	Count
EE	2
CE	1
CS	3
ECON	3
IEOR	5
stat	3
Philosophy	1
English	1
N/A	2

Engineering (6),
Financial engineering (11),
Phil. & English (2)

Programming Languages



- On average ~ 1.9 years in programming
- Most requests were on data analysis library (pandas), scientific computing, and web development

Special Topics

- Django in Python
- Some financial engineering toolkits