COMS 3101-3 Programming Languages – Python: Lecture 1

Kangkook Jee
jikk@cs.columbia.edu
Agenda

• Course description
• Introduction to Python
  - Language aspects and usage cases
• Getting started
  - How to run Python
  - Basic data types, Control flows
• Advanced data types
  - List, tuples
COURSE DESCRIPTION
Instructor

• Kangkook Jee
  – 6th year ph.d student doing security research

• Python experience
  – 4 ~ 5 years
  – Other favorite languages
    • C, C++, bash
    • Little experience with Java

• Projects done with python
  – Prototyped compiler optimizations (12 ~ 15k lines)
  – Enjoy scripting with python for everyday chores
# Syllabus

<table>
<thead>
<tr>
<th>Lecture 1</th>
<th>Python intro, set-up environments, basic data types, control flow, intro to advanced data types (list, tuples)</th>
<th>- HW1 out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lecture 2</td>
<td>More advanced data types(dictionary, string), file I/O</td>
<td>- HW1 due - HW2 out</td>
</tr>
<tr>
<td>(Sep 12)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lecture 3</td>
<td>Module and Packages, Exceptions, Object oriented programming, functional programming with lambda</td>
<td>- HW2 due</td>
</tr>
<tr>
<td>(Sep 19)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lecture 4</td>
<td>Intro to standard libraries (os, sys), serialization with pickle</td>
<td>Proposal due HW3 out</td>
</tr>
<tr>
<td>(Sep 26)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lecture 5</td>
<td>Network programming with python, multi-processing and multi-threading, debugging with pdb, python unit testing</td>
<td>- HW3 due - HW4 out</td>
</tr>
<tr>
<td>(Oct 3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lecture 6</td>
<td>Selected topics: DB programming, Web development(Django with python), Python native call, Performance optimizations</td>
<td>- HW 4 due</td>
</tr>
<tr>
<td>(Oct 10)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Special topics for lecture 6 are tentative
Logistics

• Websites
  – Piazza: / [https://piazza.com/class/hl5f5yjwj1166r](https://piazza.com/class/hl5f5yjwj1166r)

• Teaching Assistant: TBA

• Office hours
  – Tuesday 11am ~ 2pm @ CSB 504
  – Friday 11am ~ 2pm $ CSB 504
Grading / Deliverables

• Class participation: 10%

• Four homework: 40%
  – 4 homework assignments
  – Due following week before class

• Course Project
  – Project proposal: 10%
    • 1 ~ 2 pages summary/outline of course project
  – Course project deliverables: 40%
    • Leveraging python knowledge to create something interesting/useful to you

• Late policy: two grace days, after which accepted at: -10% per day
Textbooks

• No required textbook for the course
• But, some reference readings

• CLIO also has some materials available online.
Online resources

• Official Python documentation.
  – http://docs.python.org

• Official beginners guide.
  – https://wiki.python.org/moin/BeginnersGuide/Programmers

• PEP documentation.
  – http://www.peps.io

• Online Python Cookbook.
  – http://code.activestate.com/recipes/langs/python

• Tutorials
  – Official Python tutorial: http://docs.python.org/tutorial/
  – Dive into Python: http://diveintopython3.ep.io/

• More from course website.
Python Language Aspects

• Easy to learn and use
  – Clear, readable syntax
  – Large collection of standard libraries
  – Automatic memory management with *garbage collection*

• Dynamic programming language
  – Interpreted language
  – Dynamic typing
  – Introspection

• Multiple programming paradigms
  – Mainly imperative but supports functional
  – Well supported OOP

• Extensive 3\textsuperscript{rd} party modules

• Portable language
  – Different interpreters for many platforms
Scripting Language vs. Compiled Language

**Scripting Language**
- Executed from interpreter / VM
- Performance slowdown
- Type checking at runtime
- Limited functionalities
- Easy/Fast to write/debug
- Ex) shell (bash, csh), PHP, PERL

**Compiled Language**
- Executed as a native binary
- Efficient execution
- Type checking at compile time
- Advanced programming features
- Hard to debug
- Ex) C, C++, Fortan, Cobol
Python as a Scripting Language

• Shell tools
  – Launched from a console command
  – Usually, handles text inputs

• Control languages
  – Large applications exports Python API as a control front-end (IDA pro, Websphere, Sublime text)

• Development aids
  – Testing framework can be written with Python
More Use Cases

• Web development
  – With Python Djang framework
  – Yelp, YouTube, Reddit …
• Scientific / numeric computing
  – Machine learning, NLP, bioinformatics
• Complex applications with large code base
  – Dropbox, BitTorrent, Eve Online (MMORPG)
Python Deficits

• Limited support multi-threading
  – Multi-processing well supported

• Convenience comes with cost
  – Overall 10x/ 5x slowdown over programs written in C/Java
  – But, it saves your development time!
    • 3 ~ 5x less time than Java, 5 ~ 10x then C/C++
  – CPU time is cheaper than human time!
BOOTSTRAPPING PYTHON
Python Versions

• Two branches
  – Python 2
    • Current and ultimate release: 2.7
  – Python 3
    • Current latest release: 3.3.2
    • Cannot execute 2.x code

• Many important packages not (yet) ported to Python 3
• 2to3 tool exists, but does not always work
• This course: subset of Python 2.7, largely compatible with Python 3
Running Python

• Python on Linux, Mac OS X
  – Located at /usr/bin/python
  – Default version 2.7.x

• Python on Windows
  – Download an install package(2.7.x) from [http://www.python.org/download](http://www.python.org/download)
  – Execute Python.exe (C:\Python2.7\python.exe) or IDLE

• To exit: Ctrl-D (Ctrl-Z on Windows)
Two Execution Modes

Interpreter mode

- Improved shells: IDLE, bpython, ipython

Batch mode

$chmod a+x hello.py ; ./hello.py
Extended Example: hello2.py

- Import ‘sys’ module (line 2) to process command line arguments
- ‘main’ Function defined (line 5 ~ 11)
- Taking input from command line (line 7, 8)
- “__name__” variable to tell the interpreter that the script is executed from command line

Get it from http://www.cs.columbia.edu/~jikk/hello2.py

$chmod a+x hello2.py ; ./hello2.py COMS3103
Python Execution Model

• Python execution
  – Python code file names end in `.py`
  – Python interpreter executes the file from top to bottom

• Bytecode translation
  – Python first converts your sources (.py) to bytecode (.pyc)
    • Bytecode is a low-level platform independent from of your code
    • It is platform independent form and executes more quickly
  – Bytecode is executed from PVM (Python Virtual Machine)
  – If source has changed, the .py file is recompiled
Development Environments

• Text Based
  – Emacs, Vim, Sublime text

• GUI based
  – Eclipse with PyDev, Netbeans, IDLE

• Any of above are adequate for the class
  – Supports syntax highlighting, auto-completion
  – Some support: integrated debugging and code refactoring
  – Personal favorite: emacs + Ropemacs combo
LANGUAGE FUNDAMENTALS
Disclosure

• Slide Credit:
  – Daniel Bauer
  – Joshua B. Gordon

THANKS!
Elementary Python Syntax: Whitespaces Blocks

- Indentation level and line-breaks are syntactically relevant
  - Statements with same indentation belong to the same block
  - Single most hated Python feature
  - Actually useful: enforce readable code

```python
while x == 1:
    ....if y:
    .........f1()
    ....f2()
```

```c/
while (x == 1) {
    if (y) {f1();}
    f2();
}
```

- Warning: Never mix tabstops and whitespaces!
  - Do not use tabs at all (outside of strings)
  - Set your editor/IDE to fill tabs with white spaces automatically
    - Recommendation: 4 space per indentation level
Elementary Python Syntax: Linebreaks

- Python program consist of a sequence of logical lines (statements)
  - The end of physical line marks the statement
  - Statement may contain one or more physical lines by
    - Joining physical lines with “\” symbol
    - Open (, {, [ have not yet been closed, the next line joined automatically
  - Indentation level only counts after finished lines

```python
# a statement spanning multiple lines
cheeselist = ['cheddar', 'camembert', 'swiss', 'mozzarella']

# use \ to join lines
cheeselist = ['cheddar', 'camembert', 'swiss', '\
             'mozzarella']
```
Elementary Python Syntax: Comments

• Single-line comments

```python
# Print some informative messages.
print('hello world!')  # Hi there!
```

• Multi-line comments
  – Tripple “ or ‘ surrounding lines
  – Used as Docstrings at the beginning of function, method, class definitions and modules
    • For documentation with pydoc (later)

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

• Refer to style guides
  – PEP8: Official (?) style guide
  – Google’s python style guide
  – Code Like a Pythonista: Idiomatic Python

• Tools that help you with styles
  – pyflakes, pylint

• General rules
  – Limit lines to 79 characters
  – Classnames should be written in CamelCase
  – Everything else (variables, function, modules ..) should be lower_case_with_underscore
DATA TYPES AND VARIABLES
Variables and Assignments

- Evaluate expression on the right hand side of = and assign to it the variable (name) on the left hand side
- No declaration for variable is needed

```python
>>> answer
42
>>> answer += 5  # Shortcut += -= *= /=
>>> answer
47
```

- Multiple assignment in one line possible

```python
>>> a, b = 2, 3
>>> a, b = b, a  # Swap variables
>>> print a, b
3 2
```
Python Data Types: Built-In Types

- **Basic types**
  - NoneType: None
  - Bool: True, False
    - Subtype of int
  - Numerics: int (12), long (23212L), float (34.2)
- **Container types**
  - str: ‘Hello’
  - list: [1, 2, 3]
  - tuple: (1, 2, 3)
  - dict: {'A': 1, 'B':2}
  - set: {1, 2, 2, 3} → set([1, 2, 3])
- function, class, instance ...
- In Python *everything* is an object and every object has a type
  - e.g., Type object as *Type* type
Dynamic Typing

• Type checking performed at runtime
  – To Make sure variables/objects have the correct type for an operation
• No declaration needed
• Can get type of a variable with ‘type(variable)’

```python
>>> answer = 6 * 7
>>> answer += 3
>>> answer = 'fortytwo'
>>> answer += 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> type(answer)
<type 'str'>
```
Variable are Names

Objects never changes their types, but variables can be names for different objects during runtime

>>> a, b, c = 14, "fortytwo", 12.3
Variable are Names

Objects never changes their types, but variables can be names for different objects during runtime

variables  objects in memory

a  Object: 0x7f994bc10278, int 14
b  Object: 0x10b2cb600, str "fortytwo"
c  Object: 0x7f994bc129c8, float 12.3

>>> a, b, c = 14, "fortytwo", 12.3
>>> a, b = b, a
Variable are Names

Objects never changes their types, but variables can be names for different objects during runtime

```python
>>> a, b, c = 14, "fortytwo", 12.3
>>> a, b = b, a
>>> c = None
```
Variable are Names

Objects never changes their types, but variables can be names for different objects during runtime.

```
a, b, c = 14, "fortytwo", 12.3
>>> a, b = b, a
>>> c = None
```

---

variables          objects in memory

```
a
b
```

Object: 0x7f994bc10278, int 14
Object: 0x10b2cb600, str "fortytwo"

Garbage Collected
Object Mutability

• Python has mutable and immutable objects
  – Mutable objects (lists, dictionaries, sets) can be modified

```python
>>> cats = ['felix', 'dinah', 'lucky'] # list data type
>>> id(cats) # get object ID
4482450136
>>> cats.append('garfield') # add an element to the list
>>> cats
['felix', 'dinah', 'lucky', 'garfield']
>>> id(cats) # get object ID
4482450136

>>> felix = 'Felix'
>>> id(felix)
4482446800
>>> felix += 'the cat'
>>> id(felix)
4482440080
```

  – Immutable objects (boolean, numbers, string, tuple) cannot be changed once they are initialized
Python uses Strong Typing

- Operations may expect operands of certain types
- Interpreter throws an exception if type is invalid

```python
>>> answer = 6 * 7
>>> answer += 3
>>> answer = 'fortytwo'
>>> answer += 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> type(answer)
<type 'str'>
```
Evaluating Equality

- `==` for value equality
- Works for all objects (objects with different type will return `False`)

```python
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a == b
True

>>> c = 1
>>> d = '1'
>>> c == d
False
```

- ‘`is’` for object equality
  - if two variables are names for the same object

```python
>>> a is b
False
>>> id(a) == id(b)
False
```
Comparison Operators

• All comparison operators work for all objects
  – Value equality: ==, !=, <, <=, >, >=
  – Object equality: is, is not
• Comparison operators return an object of type bool (True, False)
• Result of comparison can be combined with boolean operators
  – not x
  – x and y
  – x or y

```python
>>> a, b = 5, 7
>>> a >= 6
False
>>> False or (a == 5)
True
>>> a > 0 and not False
True
```
Numeric Operators

- **Binary**: \(x + y, x - y, x \times y, x \div y, x^{y}\) (power), \(x \mod y\) (modulo)
- **Unary**: \(+x, -y, \text{not } x\)
- **Built-in functions**
  - Convert to int / long / float: `int(x)`, `long(x)`, `float(x)`
  - Absolute value: `abs(x)`

```python
>>> 20 / 3
6
>> 20 % 3
2
>>> float(20) / 3  # type conversion
6.666666666666667
>>> float("1213")  # can convert(parse) string
1213.0
```
CONTROL FLOWS
Conditionals: if Statement

- If one `if` or `elif` matches the *indented* block statement is executed
  - Remaining conditions are ignored
- `elif` and `else` are optional
- If no `if` or `elif` is matches, the indented block statement for `else` is executed
- There is no `switch` statement in Python

```python
if conditionExp1:
    statement1
    ...
elif conditionExp2:
    statement2
    ...
elif conditionExp3:
    statement3
    ...
else:
    statement4
    ...
```
Expressions in if and elif Conditions

• Can use any expression as a condition
  – Will be casted to boolean type
  – 0 (number), None, empty containers (string, list, tuple, dict, set) \( \rightarrow \) False
  – Any object \( \rightarrow \) True

• Use boolean operators (and, or) to combine multiple objects
Loops: while Statements

- Execute the indented statements repeatedly while conditionExp evaluates True
- else branch is visited loop terminates as conditionExp being False

```python
count = 0
while x > 0:
    x=x/2
    count += 1
else:
    print('approximate log2:')
    print(count)
```
continue and break

‘continue’ interrupts the current iteration of the loop and continues at the next iteration.

‘break’ interrupts the complete loop and escape to statements below the loop.

```python
>>> x = 5
>>> while x:
...     x -= 1
...     if not x % 2:
...         continue
...     print (x)
...
3
1
```

```python
>>> x = 10
>>> while True:
...     print (x)
...     x -= 1
...     if x == 7:
...         break
...
10
9
8
```
Loops: for Statements

- Python’s for statement iterates over the items of any sequence (a list or a string), in order that appears from the sequence.
- "else" branch is visited when loop exhaust all entries from sequence.

```python
>>> sentence=""
>>> for word in ["hello! ", "COMS3101", ",-3"]:  
...     sentence += word  
... else:  
...     print (sentence)  
...  
hello! COMS3101-3
```
ADVANCED TYPES
Sequence Types

• Container objects that contain ordered sequences of elements:
  – String (a sequence of encoded characters)
    \[ x = 'Read me! I’m string!' \]
  – list (mutable sequence of objects)
    \[ x = [4, 8, 9, 10] \]
  – tuple (mutable sequence of objects)
    \[ x = (10, 12, “hello”) \]

• All sequence types supports some common operations
  – Get length, concatenation and repetition
  – Test for membership
  – Access for specific elements and ‘slicing’
  – Iterate through elements
Length, Concatenation and Repetition

- `len(x)` returns the length of sequence `x`

  ```
  >>> x = []  # empty list
  >>> len(x)
  0
  >>> len("number of characters in string")
  30
  ```

- `x + y` concatenates sequence of `x` and `y`

  ```
  >>> 'hello' + 'COMS3101'
  'helloCOMS3101'
  ```

- `x * n` or `n * x` repeats sequence `x` for `n` times

  ```
  >>> 3 * ('rep',)  # single entry tuple
  ('rep', 'rep', 'rep')
  ```
Testing for Sequence Membership

• \texttt{x in y} returns \texttt{True} if collection \texttt{y} contains object \texttt{x}, False otherwise
  – Based on value equality (==)
  – \texttt{x not in y} is equivalent to not \texttt{x in y}

\begin{Verbatim}
>>> 'coffee' in ['tea', 'coffee', 'juice']
True
\end{Verbatim}

• For string only:
  – \texttt{in} also tests if \texttt{x} is a substring of \texttt{y}

\begin{Verbatim}
>>> 'tuna' in 'fortunate'
True
\end{Verbatim}
Finding Index and Counting Element

- `x.index(y)` returns the sequence index of the first occurrence of `y`

```python
>>> (23, 5, 8, 5).index(5)
1
```

- `x.count(y)` returns the number of times `y` occurs in `x`

```python
>>> 'banana'.count('a')
3
>>> 'banana'.count('an')  # works for substring
2
```
Sequence Indexing

• x[i] indexes the element of sequence x (starting from 0)

```python
>>> x = ((1, 2, 3), 'foo', 1.0)
>>> x[1]
'foo'
>>> x[0][2]  # nested indexing
3
```

• reverse indexing starts at -1

```python
>>> x = ((1, 2, 3), 'foo', 1.0)
>>> x[-1]
1.0
```
Sequence Slicing

• Slicing returns a copy of subsequence
• $x[i:j]$ returns the subsequence from position $i$ (inclusive) to position $j$ (exclusive)
• $x[i:]$ returns the subsequence from position $i$ from to the end
• $x[:j]$ returns the subsequence from the beginning to position $j$ (exclusive)

```python
>>> x = [0,1,2,3,4]
>>> x[1:]
[1,2,3,4]
>>> x[:-2]  # using reverse indexing in slice indices
[0,1,2]
>>> x[2:3]
[2]
```
Iterating Elements Through Sequence

• Sequence data types implements the iterator protocol
• For-loop can *iterate* any sequence types

```python
>>> sentence=""
>>> for word in ["hello! ", "COMS3101", ","-3"]:
...     sentence += word
... else:
...     print (sentence)
...
hello! COMS3101-3
```