A Brief Introduction to Python for those who know Java

Last extensive revision:
   Jie Gao, Fall 2019
Previous revisions:
   Daniel Moroz, Fall 2015
Plan Day 1

- Baby steps
  - History, Python environments, Docs
- Absolute Fundamentals
  - Objects, Types
  - Math and Strings basics
  - References and Mutability
- Data Types
  - Strings, Tuples, Lists, Dicts
- Looping
  - Comprehensions
- Iterators
  - Generators
- To Be Continued…
Python

- Developed by Guido van Rossum in the early 90s
  - Originally Dutch, in USA since 1995, now works for Dropbox
  - Benevolent Dictator for Life (has now stepped down)
- Available on Eniac; download at python.org
- Named after the Monty Python comedy group
Some Positive Features of Python

- **Fast development:**
  - Concise, intuitive syntax
    - Whitespace delimited
  - Garbage collected

- **Portable:**
  - Programs run on major platforms without change
  - cpython: common Python implementation in C.

- **Various built-in types:**
  - lists, dictionaries, sets: useful for AI

- **Large collection of support libraries:**
  - NumPy for Matlab like programming
  - Sklearn for machine learning
  - Pandas for data analysis
Recommended Reading

- **Python Overview**
  - The Official Python Tutorial ([https://docs.python.org/3/tutorial/index.html](https://docs.python.org/3/tutorial/index.html))
  - Slides for CIS192, Spring 2019 ([https://www.cis.upenn.edu/~cis192/](https://www.cis.upenn.edu/~cis192/))

- **PEPs – Python Enhancement Proposals**
  - PEP 8 - Official Style Guide for Python Code (Guido et al)
    - Style is about consistency. 4 space indents, < 80 char lines
    - Naming convention for functions and variables: lower_w_under
    - Use the automatic pep8 checker!
  - PEP 20 – The Zen of Python (Tim Peters) (try: `import this`)
    - Beautiful is better than ugly; simple is better than complex
    - There should be one obvious way to do it
    - That way may not be obvious at first unless you're Dutch
    - Readability counts
Which Python?

- Python 3.7.0 is the latest stable version
  - Current version on Eniac is 3.6.5, so we’ll use it
  - Last stable release before version 3.7.4

- Why not Python 2?
  - Latest version 2.7.15.
  - Python 2.7 will no longer be supported after 2020 (https://pythonclock.org/)
  - Reading: What should I learn as a beginner: Python 2 or Python 3?
Python REPL Environment

- **REPL**
  - Read-Evaluate-Print Loop
  - Type “python” at the terminal
  - Convenient for testing
  - If you’d like syntax highlighting in REPL try `bpython`

```bash
$cis521x@eniac:~$ python3
Python 3.4.6 (default, Mar 22 2017, 12:26:13) [GCC] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello World!')
Hello World!
>>> 'Hello World!'
'Hello World!'
>>> [2*i for i in range(10)]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> exit()
$cis521x@eniac:~$  
```
Python Scripts

- Scripts
  - Create a file with your favorite text editor (like Sublime)
  - Type “python3 script_name.py” at the terminal to run
  - Not REPL, need to explicitly print
  - Homework submitted as scripts

```python
import random
def rand_fn():
    """outputs list of 10 random floats between [0.0, 1.0)""
    return ["%.2f" % random.random() for i in range(10)]

print('1/2 = ', 1/2)
if __name__ == '__main__':
    rand_fn()
    print(rand_fn())

$cis521x@eniac:~> python3 foo.py$
1/2 = 0.5
['0.08', '0.10', '0.84', '0.01', '0.00', '0.59', '0.67', '0.88', '0.58', '0.81']
```
PyCharm IDE
Python Notebooks

- Jupyter Notebooks and Google colab allow you to interactively run Python code in your web browser and share it with others.
- They are popular for tutorials since you can include inline text and images.
Structure of Python File

- **Whitespace is meaningful in Python**
- **Use a newline to end a line of code.**
  - Use \ when must go to next line prematurely.
- **Block structure is indicated by indentation**
  - The first line with less indentation is outside of the block.
  - The first line with more indentation starts a nested block.
  - Often a colon appears at the end of the line of a start of a new block. (E.g. for function and class definitions.)
Objects and Types

- **All data treated as objects**
  - An object is deleted (by garbage collection) once unreachable.

- **Strong Typing**
  - Every object has a fixed type, interpreter doesn’t allow things incompatible with that type (e.g. “foo” + 2)
  - `type(object)`
  - `isinstance(object, type)`

- **Examples of Types:**
  - `int`, `float`
  - `str`, `tuple`, `dict`, `list`
  - `bool`: True, False
  - `None`, `generator`, `function`
Static vs Dynamic Typing

- **Java: static typing**
  - Variables can only refer to objects of a declared type
  - Methods use type signatures to enforce contracts

- **Python: dynamic typing**
  - Variables come into existence when first assigned.
    - `>>> x = "foo"
    - `>>> x = 2`
  - `type(var)` automatically determined
  - If assigned again, `type(var)` is updated
  - *Functions have no type signatures*
  - Drawback: type errors are only caught at runtime
Math Basics

- **Literals**
  - Integers: 1, 2
  - Floats: 1.0, 2e10
  - Boolean: True, False

- **Operations**
  - Arithmetic: + - * /
  - Power: **
  - Modulus: %
  - Comparison: , <=, >=, ==, !=
  - Logic: (and, or, not) *not symbols*

- **Assignment Operators**
  - += *= /= &= ...
  - No ++ or --
Strings

- **Creation**
  - Can use either single or double quotes
  - Triple quote for multiline string and docstring

- **Concatenating strings**
  - By separating string literals with whitespace
  - Special use of ‘+’

- **Prefixing with r means raw.**
  - No need to escape special characters: r"\n"

- **String formatting**
  - Special use of ‘%’ (as in printf in C)
  - `print("%s can speak %d languages" % ("C3PO", 6000000))`
A Simple Code Sample

```python
x = 34 - 23  # A comment.
y = 'Hello'   # Another one.
z = 3.45
if z == 3.45 or y == 'Hello':
    x = x + 1
    y = y + ' World'  # String concat.
print(x)
print(y)
```
## References and Mutability

```python
>>> x = 'foo'
>>> y = x
>>> x = x.strip() # new obj
>>> x
'foo'
>>> y
'foo'
```

- **Strings are immutable**
- `==` checks whether variables point to objects of the same value
- `is` checks whether variables point to the same object

```python
>>> x = [1, 2, 3]
>>> y = x
>>> x.append(5) # same obj
>>> y
[1, 2, 3, 5]
>>> x
[1, 2, 3, 5]
```

- **Lists are mutable**
- Use `y = x[:]` to get a (shallow) copy of any sequence, ie. a new object of the same value
Sequence types:
Tuples, Lists, and Strings
Sequence Types

- **Tuple**
  - A simple *immutable* ordered sequence of items
    - *Immutable*: a tuple cannot be modified once created
  - Items can be of mixed types, including collection types

- **Strings**
  - *Immutable*
  - Very much like a tuple with different syntax
  - Regular strings are Unicode and use 2-byte characters
    (Regular strings in Python 2 use 8-bit characters)

- **List**
  - *Mutable* ordered sequence of items of mixed types
Sequence Types

- The three sequence types share much of the same syntax and functionality.

```python
>>> tu = (23, 'abc', 4.56, (2,3), 'def')  # tuple
```

```python
>>> li = ['abc', 34, 4.34, 23]  # list
```

```python
>>> st = "Hello World"; st = 'Hello World'  # strings
```

```python
>>> tu[1]  # Accessing second item in the tuple.
'abc'
```

```python
>>> tu[-3]  # negative lookup from right, from -1
4.56
```
Slicing: Return Copy of a Subsequence

>>> t = (23, 'abc', 4.56, (2,3), 'def')

>>> t[1:4] # slicing ends before last index
('abc', 4.56, (2,3))

>>> t[1:-1] # using negative index
('abc', 4.56, (2,3))

>>> t[1:-1:2] # selection of every nth item.
('abc', (2,3))

>>> t[:2] # copy from beginning of sequence
(23, 'abc')

>>> t[2:] # copy to the very end of the sequence
(4.56, (2,3), 'def')
Operations on Lists

```python
>>> li = [1, 11, 3, 4, 5]
>>> li.append('a')  # Note the method syntax
>>> li
[1, 11, 3, 4, 5, 'a']
>>> li.insert(2, 'i')
>>> li
[1, 11, 'i', 3, 4, 5, 'a']
>>> li = ['a', 'b', 'c', 'b']
>>> li.index('b')  # index of first occurrence
1
>>> li.count('b')  # number of occurrences
2
>>> li.remove('b')  # remove first occurrence
>>> li
['a', 'c', 'b']
```
Operations on Lists 2

```python
>>> li = [5, 2, 6, 8]

>>> li.reverse()  # reverse the list *in place* (modify)
>>> li
[8, 6, 2, 5]

>>> li.sort()  # sort the list *in place*
>>> li
[2, 5, 6, 8]

>>> li.sort(some_function)
# sort in place using user-defined comparison

>>> sorted(li)  # return a *copy* sorted
```

Operations on Strings

```python
>>> s = "Pretend this sentence makes sense."
>>> words = s.split(" ")
>>> words
['Pretend', 'this', 'sentence', 'makes', 'sense. ']
>>> ".join(words) #join method of obj ""
'Pretend_this_sentence_makes_sense.'

>>> s = 'dog'
>>> s.capitalize()
'Dog'
>>> s.upper()
'DOG'
>>> ' hi --'.strip(' - ')
'hi'
```

[https://docs.python.org/3.7/library/string.html](https://docs.python.org/3.7/library/string.html)
Tuples

```python
>>> a = ['apple', 'orange', 'banana']
>>> for (index, fruit) in enumerate(a):
...     print(str(index) + ': ' + fruit)
...
0: apple
1: orange
2: banana
```

```python
>>> a = [1, 2, 3]
>>> b = ['a', 'b', 'c', 'd']
>>> list(zip(a, b))
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> list(zip("foo", "bar"))
[('f', 'b'), ('o', 'a'), ('o', 'r')]
>>> x, y, z = 'a', 'b', 'c'
```
Dictionaries: a *mapping* collection type
Dict: Create, Access, Update

- Dictionaries are unordered & work by hashing, so keys must be immutable
- Constant average time add, lookup, update

```python
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user']
'bozo'

>>> d['bozo']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'bozo'

>>> d['user'] = 'clown'  # Assigning to an existing key replaces its value.

>>> d
{'user': 'clown', 'pswd': 1234}
```
Dict: Useful Methods

```python
>>> d = {'user': 'bozo', 'p': 1234, 'i': 34}
>>> d.keys()  # List of current keys
dict_keys(['user', 'p', 'i'])
>>> d.values()  # List of current values.
dict_values(['bozo', 1234, 34])
>>> d.items()  # List of item tuples.
dict_items([('user', 'bozo'), ('p', 1234), ('i', 34)])

>>> from collections import defaultdict
>>> d = defaultdict(int)
>>> d['a']
0
```

- `defaultdict` automatically initializes nonexistent dictionary values
For Loops
For Loops

```
for <item> in <collection>:
    <statements>
```

- If you’ve got an existing list, this iterates each item in it.
- You can generate a list with `Range`:
  - `list(range(5))` returns `[0,1,2,3,4]`
  - So we can say:
    ```
    for x in range(5):
        print(x)
    ```
- `<item>` can be more complex than a single variable name.
  ```
  for (x, y) in [('a',1), ('b',2), ('c',3), ('d',4)]:
      print(x)
  ```
List Comprehensions replace loops!

nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# I want 'n*n' for each 'n' in nums
squares = []
for n in nums:
    for n in nums:
        squares.append(x*x)
print(squares)

squares = [x*x for x in nums]
print(squares)
List Comprehensions replace loops!

```python
>>> li = [3, 6, 2, 7]
>>> [elem * 2 for elem in li]
[6, 12, 4, 14]

>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [n * 3 for (x, n) in li]
[3, 6, 21]
```
Filtered List Comprehensions

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem * 2 for elem in li if elem > 4]
[12, 14, 18]
```

- Only 6, 7, and 9 satisfy the filter condition.
- So, only 12, 14, and 18 are produced.
List Comprehension extra for

\[
\text{lst1, lst2, lst3} = [1, 2, 3], [2, 3, 4], [3, 4, 5]
\]

\[
\text{res} = [(x, y, z) \text{ for } x \text{ in } \text{lst1} \text{ if } x < 2 \ \text{for } y \text{ in } \text{lst2} \ \text{for } z \text{ in } \text{lst3} \text{ if } x + y + z < 8]
\]

\[
\text{res} = [] \ # \ text{translation}
\]
\[
\text{for } x \text{ in } \text{lst1}:
\]
\[
\quad \text{if } x < 2:
\]
\[
\quad \quad \text{for } y \text{ in } \text{lst2}:
\]
\[
\quad \quad \quad \text{for } z \text{ in } \text{lst3}:
\]
\[
\quad \quad \quad \quad \text{if } x + y + z < 8:
\]
\[
\quad \quad \quad \quad \quad \text{res}.\text{append}((x, y, z))
\]

# Both value of res: [(1, 2, 3), (1, 2, 4), (1, 3, 3)]
Dictionary, Set Comprehensions

```python
lst1 = [('a', 1), ('b', 2), ('c', 'hi')]
lst2 = ['x', 'a', 6]

D = {k: v for k, v in lst1}
s = {x for x in lst2}

D = dict()  # translation
for k, v in lst1:
    d[k] = v
s = set()  # translation
for x in lst:
    s.add(x)

# Both value of d: {'a': 1, 'b': 2, 'c': 'hi'}
# Both value of d: {'x', 'a', 6}'
```
Iterators
Iterator Objects

- Iterable objects can be used in a for loop because they have an `__iter__` magic method, which converts them to iterator objects:

  >>> k = [1,2,3]
  >>> k.__iter__()
  <list_iterator object at 0x104f8ca50>

  >>> iter(k)
  <list_iterator object at 0x104f8ca10>
Iterators

- Iterators are objects with a \_\_next\_\_() method:
  >>> i = iter(k)
  >>> next(i)
  1
  >>> i.__next__()
  2
  >>> i.next()
  3
  >>> i.next()
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  StopIteration

- Python iterators do not have a hasnext() method!
- Just catch the StopIteration exception
Iterators: The real truth about `For.. In..`

- for `<item>` in `<iterable>`:
  - `<statements>`

- First line is just syntactic sugar for:
  - 1. Initialize: Call `<iterable>.__iter__()` to create an `iterator`
    Each iteration:
    - 2. Call `iterator.__next__()` and bind `<item>`
    - 2a. Catch `StopIteration` exceptions

- To be iterable: has `__iter__` method
  - which returns an iterator obj

- To be iterator: has `__next__` method
  - which throws `StopIteration` when done
class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

    def __iter__(self):
        return self

>>> for char in Reverse('spam'):
    print(char)

m
a
p
s
Iterators use memory efficiently

Eg: File Objects

```python
>>> for line in open("script.py"):  # returns iterator
...     print(line.upper())

...     print(line.upper())
...

import sys
print(sys.path)
x = 2
print(2 ** 3)
```

instead of

```python
>>> for line in open("script.py").readlines():  # returns list
...     print(line.upper())
...
```
Generators
Generators: using `yield`

- Generators are iterators (with `__next__` method)
- Creating Generators: `yield`
  - Functions that contain the `yield` keyword `automatically` return a generator when called

```python
>>> def f(n):
...    yield n
...    yield n+1
...  
...  
>>> type(f)
<class 'function'>
>>> type(f(5))
<class 'generator'>
>>> [i for i in f(6)]
[6, 7]
```
Generators: What does `yield` do?

- Each time we call the `__next__` method of the generator, the method runs until it encounters a `yield` statement, and then it stops and returns the value that was yielded. Next time, it resumes where it left off.

```python
>>> gen = f(5) # no need to say f(5).__iter__()
>>> gen
<generator object f at 0x1008cc9b0>
>>> gen.__next__()
5
>>> next(gen)
6
>>> gen.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```
Generators

- **xrange(n) vs range(n) in Python 2**
  - **xrange** acts like a generator
  - **range(n)** keeps all n values in memory before starting a loop even if n is huge: for k in range(n)
  - `sum(xrange(n))` much faster than `sum(range(n))` for large n

- **In Python 3**
  - **xrange(n) is removed**
  - **range(n) acts similar to the old xrange(n)**
  - Can use list() to get similar behavior as in Python 2
  - Python 3’s range is more powerful than Python 2’s xrange
Generators

- **Benefits of using generators**
  - Less code than writing a standard iterator
  - Maintains local state automatically
  - Values are computed one at a time, as they’re needed
  - Avoids storing the entire sequence in memory
  - Good for aggregating (summing, counting) items. One pass.
  - Crucial for infinite sequences
  - Bad if you need to inspect the individual values
Using generators: merging sequences

- Problem: merge two sorted lists, using the output as a stream (i.e. not storing it).

```python
def merge(l, r):
    llen, rlen, i, j = len(l), len(r), 0, 0
    while i < llen or j < rlen:
        if j == rlen or (i < llen and l[i] < r[j]):
            yield l[i]
            i += 1
        else:
            yield r[j]
            j += 1
```
Using generators

```python
>>> g = merge([2,4], [1, 3, 5]) # g is an iterator
>>> while True:
...    print(g.__next__())
...
1
2
3
4
5
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
StopIteration

>>> [x for x in merge([1,3,5],[2,4])]
[1, 2, 3, 4, 5]
```
Generators and exceptions

```python
>>> g = merge([2,4], [1, 3, 5])
>>> while True:
...     try:
...         print(g.__next__())
...     except StopIteration:
...         print('Done')
...         break

1
2
3
4
5
Done
```
Generator comprehensions

- **Review**: generators are good for aggregating items.

- **For example, in Python 2**, \( \text{sum(xrange(n))} \) was *much faster than* \( \text{sum(range(n))} \) *for large n*

- **Similarly**,

  ```python
  >>> sum(x for x in xrange(10**8) if x%5==0)
  999999950000000L
  which uses a *generator comprehension* is much faster than
  ```

  ```python
  >>> sum([x for x in xrange(10**8) if x%5==0])
  999999950000000L
  which creates the entire list before computing the sum
  ```
Plan for next time

- **Import**
- **Functions**
  - Args, kwargs
- **Classes**
  - “magic” methods (objects behave like built-in types)
- **Profiling**
  - timeit
  - cProfile