The Python Tutorial

Deep Learning and Practice @ MediaTek Inc.

Department of Computer Science, NCTU

Introduction

- Python is simple to use, but it is a real programming language, offering much more structure and support for large programs than shell scripts can offer.
- Python is an interpreted language, which can save you considerable time during program development because no compilation and linking is necessary.

  The interpreter can be used interactively, which makes it easy to experiment with features of the language, or to test functions during bottom-up program development.
- Nowadays, the top deep learning frameworks or libraries are available on the Python.

  PyTorch, TensorFlow, Caffe, Keras, Theano, etc.

Whetting Your Appetite

- What you will learn in this tutorial:
  - Basic concepts and features of the Python language.
  - The usage of common data structures and APIs.
  - How to use the Python interpreter and scripts.
  - Some simple examples written in Python.
- What you will not learn in this tutorial:
  - Documentation of the Python language.
  - Detailed explanation of Python's syntax and semantics.
  - Theory or complex algorithms of deep learning.
  - The usage of PyTroch or any other frameworks.

Using the Python Interpreter

- The Python interpreter is usually installed in `/usr/bin/` or `/usr/local/bin/` on those machines where it is available.
- To start the interpreter, type the following commands in the terminal:

```
$ python3     # to start Python 3
$ python2     # to start Python 2
$ python3.6   # specify the version of Python
```

- Python 2 and Python 3 are NOT fully compatible. You should avoid using Python without the version:

```
$ python      # do not do this
```

Exercise 1-1: Using Python

- Launch your laptop, login to the workstation.

  Ask TAs for help if you do not have the login information.

- Once you are logged in, try the following command.

```
$ python3  # to start Python 3
```

- You will see the version of Python, and the prompt.

```
Python 3.6.1 (default, Mar 27 2017, 00:27:06)
[GCC 6.3.1 20170306] on linux
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

- Type the following command to see a “Hello, World”:

```
>>> print("Hello, World")
```

- You can use `exit()` or `Ctrl+D` to exit the interpreter.

```
>>> exit()
```

Using Python Scripts

- Python scripts usually named with `.py` suffix.
- You can execute a script by following commands:

```
$ python3 hello.py  # to start the script in Python 3
$ python2 hello.py  # to start the script in Python 2
```

- On Unix systems, Python scripts can be made directly executable by adding a Unix “Shebang” line.

```
$ cat hello.py
#!/usr/bin/env python3
print("Hello, World")

$ chmod +x hello.py # set the executable permission
$ ./hello.py        # run the script
```

Exercise 1-2: Python Scripts

- Write a "`hello.py`", execute it on the workstation.
  Try both "`python3 hello.py`" and "`./hello.py`".
- You can either create the script in the terminal directly, or edit it on your local computer and then upload it to the workstation.
  To edit in the terminal, you may want to try vim or emacs.
  To upload the script via SFTP, try FileZilla or WinSCP.
- Remember not to add non-ASCII characters into your script, otherwise you should save the script in Unicode and explicitly clarify it.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

Numbers

- The interpreter acts as a simple calculator.

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5          # division always returns a floating point
1.6
>>> 4 * 3.75 - 1  # full support for floating point
14.0
>>> 17 % 3
2
```

- Some special operators.

```
>>> 17 // 3  # floor division discards the fractional part
5
>>> 2 ** 7   # 2 to the power of 7
128
```

Using the Variables

- The equal sign (=) is used to assign a value to a variable:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height # no equal sign, so the result is displayed
900
>>> length, depth = 10, 20 # multiple assignment is possible
```

- You do not need to define the variable with its type.

```
>>> n = 0
>>> type(n) # check the object type
<class 'int'>
>>> n = 0.0 # reuse variable “n” to store another type
>>> type(n)
<class 'float'>
```

- If a variable is not “defined” (assigned a value), trying to use it will give you an error.

Exercise 2-1: Assignment

- In Python, you may assign several variables with a single equal sign.

```
>>> length, depth = 10, 20 # multiple assignment
```

- Type the following commands into the interpreter and see what happens:

```
>>> a, b = 0, 1; print(b)
>>> a, b = b, a + b; print(b)
>>> a, b = b, a + b; print(b)
>>> a, b = b, a + b; print(b)
>>> a, b = b, a + b; print(b)
>>> ...
```

- Note that each line contains two statements: an assignment and a print function

Strings

- Strings can be enclosed in `'single quotes'` or `"double quotes"` with the same result.
- Python strings cannot be changed (immutable).
- Some useful operations of strings:

```
>>> 3 * 'un' + 'ium' # 3 times 'un', followed by 'ium'
'unununium'
>>> 'Py' 'thon'      # string literals are concatenated
'Python'
>>> prefix = 'Py'
>>> s = prefix + 'thon'        # concatenate variables
>>> n = len(s)                 # the length of a string
>>> "the length is " + str(n) # converts to strings
'the length is 6'
```

Strings

- String literals can span multiple lines:

```
print("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

- The `print()` function produces a more readable output.

```
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
>>> print('"Isn\'t," she said.')
"Isn't," she said.

>>> s = 'First line.\nSecond line.'
>>> s                 # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a new line
First line.
Second line.
```

Strings Indexing

- Strings can be indexed (subscripted), with the first character having index 0.

```
>>> word = 'Python'
>>> word[0] # character in position 0
'P'
>>> word[5] # character in position 5
'n'
```

- Indices may also be negative numbers, to start counting from the right.

```
>>> word[-1] # last character
'n'
>>> word[-2] # second-last character
'o'
>>> word[-6]
'P'
```

Strings Slicing

- In addition to indexing, slicing is also supported.

```
>>> word[0:2] # from position 0 to 2 (excluded)
'Py'
>>> word[2:5] # from position 2 to 5 (excluded)
'tho'
```

- Slice indices have useful defaults.

```
>>> word[:2]   # from the beginning to position 2
'Py'
>>> word[4:]   # from position 4 to the end
'on'
>>> word[-2:] # from the second-last to the end
'on'
```

- The indexing table:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Exercise 2-2: Output Formatting

- Data can be printed in a human-readable form.

```
>>> print("The total is", 100)
>>> print(str(10).rjust(4))
>>> print("Hello,", end=' ')
>>> print("World")
>>> '-3.14'.zfill(7)
>>> '3.14159265359'.zfill(5)
>>> print('{} is {} to {}!'.format(
... 'format', 'easy', 'use'))
>>> print('{1} and {0}'.format('bar', 'foo'))
>>> print('{0:2d} {1:3d} {2:4d}'.format(4, 16, 64))
>>> print('The {object} is {adjective}.'.\
... format(object='lesson', adjective='nice'))
>>> print('%s is approx %.2f.' % ("PI", 3.1416))
```

Lists

- Lists can be written as a list of comma-separated values (items) between square brackets.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
>>> data = ['Hello', 10, 3.1416] # different types
```

- All built-in sequence types can be indexed and sliced.

```
>>> squares[-3:] # slicing returns a new list
[9, 16, 25]
```

- Lists support concatenation or multiplication.

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> [0, 1, 2] * 3
[0, 1, 2, 0, 1, 2, 0, 1, 2]
```

Lists

- Lists are a mutable type, it is possible to change their content.

```
>>> cubes = [1, 8, 27, 65, 125] # something's wrong here
>>> cubes[3] = 4 ** 3           # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

- Assignment to slices is also possible.

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters[2:5] = ['C', 'D', 'E'] # replace some values
>>> letters[2:5] = []              # now remove them
>>> letters[:] = []                # clear the list
```

- It is possible to nest lists.

```
>>> a, n = ['a', 'b', 'c'], [1, 2, 3]
>>> x = [a, n] # now x is [['a', 'b', 'c'], [1, 2, 3]]
```

Exercise 2-3: List Manipulation

- The list data type has some more methods.

```
>>> L = [0, 1, 2, 0]
>>> L.append(3)     # equivalent to L = L + [3]
>>> L.extend(L)     # also try L.append(L)
>>> L.insert(0, 5) # insert 5 at index 0
>>> L.remove(0)     # note that 0 is NOT index
>>> L.pop(4)        # note that 4 is index
>>> L.index(2)      # also try L.index(2, 4)
>>> L.count(1)
>>> L.sort()
>>> L.reverse()
>>> L.copy()
>>> L.clear()
```

Tuples and Sequences

- A tuple consists of a number of values separated by commas.

```
>>> t = 12345, 54321, 'hello!'
>>> type(t)
<class 'tuple'>
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
```

- Tuples may be nested:

```
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Tuples and Sequences

- Tuples are immutable, but they can contain mutable objects.

```
>>> t = 12345, 54321, 'hello!'
>>> t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v[0][0] = 88888
```

- All built-in sequence types support unpacking:

```
>>> t = [12345, 54321], 'hello!'
>>> l, s = t                  # unpack from a tuple
>>> n, m = l                  # unpack from a list
>>> a, b, c, d, e, f = s      # unpack from a string
>>> (n, m), (a, b, c, d, e, f) = t
```

Exercise 2-4: Circular Reference

- Execute the following commands in order:

```
>>> data = 'Hello!'
>>> data = data, 12345
>>> data = [ data, data ]
>>> data = [ data, (data, 54321) ]
>>> data[0] = data
>>> data[1][0][1] = data
```

- What is `data` now?
  What about `data[0][1][0][1][0]`?

  You can find the identity of an object by `id()`.

  Different objects are guaranteed to have different identity.

- How to obtain 12345 by unpacking?

if Statements

- The `if` statement is used for conditional execution.

```
>>> lr = float(input("Enter the learning rate: "))
Enter the learning rate: 0.5
>>> if lr < 0:
...     lr = 0
...     print('Negative changed to zero')
... elif lr == 0:
...     print('Zero')
... elif lr <= 0.5:
...     print('Fine')
... else:
...     print('Too large')
...
Fine
```

- You can use logical operations: keyword and, or, `is`, not.

```
>>> if lr > 0.25 and lr < 0.75:
...     pass
```

if Statements

- The body of the statement is indented.
  - Indentation is Python's way of grouping statements.
  - At the interactive prompt, you have to type a tab for each indented line.
  - In practice you should prepare more complicated input for Python with a text editor.
- An `if ... elif ... elif ...` sequence is a substitute for the switch or case statements found in other languages.
- Ternary operator is available on assignments, which is equivalent to `(x ? y : z)` in C or other languages:

```
>>> n = int(input("n = "))
>>> m = n ** 2 if n < 100 else n * 2
```

Exercise 3-1: Leap Year

- Determine a specific year is leap year or not.
  The pseudocode of leap year is described as follows:

  if (year is not divisible by 4) then (it is a common year)
  else if (year is not divisible by 100) then (it is a leap year)
  else if (year is not divisible by 400) then (it is a common year)
  else (it is a leap year)

- Write a "`leapyear.py`", read a user input and output "XXXX is Leap Year" or "XXXX is Common Year"

```
$ cat leapyear.py
#!/usr/bin/env python3
year = int(input("Please enter a year: "))
pass # Your solution here
```

for Statements

- The `for` statement in Python differs a bit from what you may be used to in C or Pascal.

```
>>> words = ['cat', 'window', 'defenestrate']
>>> for w in words: # iterates over the items
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

- Make a copy first if you need to modify the sequence you are iterating over while inside the loop:

```
>>> for w in words.copy():
...     if len(w) > 6:
...         words.remove(w)
```

for Statements

- If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy.

```
>>> for i in range(10):
...     print(i)
```

- You can specify the start point, end point, or the step of `range()`:

```
range(10)
   0 through 9
range(5, 10)
   5 through 9
range(0, 10, 3)
   0, 3, 6, 9
range(-10, -100, -30)
  -10, -40, -70
```

- `break` and `continue` Statements are available in Python.

Exercise 3-2: Diamond

- Given the rows (an odd number) of the diamond.
  A diamond of 5 rows should be printed as:

```
  *
 ***
*****
 ***
  *
```

- Write a "`diamond.py`", print the diamond of stars.

```
$ cat diamond.py
#!/usr/bin/env python3
n = int(input("Size of the Diamond: "))
pass # Your solution here
```

for Statements

- To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```
>>> a = ['I', 'have', 'a', 'dream']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 I
1 have
2 a
3 dream
```

- However, you should use the `enumerate()` in Python.

```
>>> list(enumerate(a))
[(0, 'I'), (1, 'have'), (2, 'a'), (3, 'dream')]
>>> for i, w in enumerate(a):
...     print(i, w)
...
```

for Statements

- To loop over two or more sequences at the same time, the entries can be paired with the `zip()` function.

```
>>> name = ['Lab 1', 'Lab 2', 'Lab 3']
>>> date = ['10/25', '11/08', '11/22']
>>> for n, d in zip(name, date):
...     print('{}: {}'.format(n, d))
...
Lab 1: 10/25
Lab 2: 11/08
Lab 3: 11/22
```

- To loop over a sequence in reversed (or sorted) order, use the `reverse()` (or `sorted()`) function.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
```

while Statements

- The structure of a `while` loop is simple:

```
>>> n = 0
>>> while n < 4:
...     print('n is', n)
...     n = n + 1
...
n is 0
n is 1
n is 2
n is 3
```

- Note that the above example works.
  However, you should use `for` statement and `range()` to iterate 0 through 3 in Python.

Exercise 3-3: Fibonacci Series

- Write a python script "`fibonacci.py`", produce the first 20 Fibonacci numbers with a `while` loop and store them into a list.

```
$ cat fibonacci.py
#!/usr/bin/env python3
fib = [ 0, 1 ]
while len(fib) < 20:
    pass  # Your solution here
print(fib)
```

- The output should be `[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]`

List Comprehensions

- List comprehensions provide a concise way to create lists.
- For example, assume we want to create a list of squares, like:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- We can calculate the list of squares without any side effects using:

```
>>> squares = [x**2 for x in range(10)]
```

List Comprehensions

- Another example, combines the elements of two lists if they are not equal:

```
>>> X, Y = [1,2,3], [3,1]
>>> combs = []
>>> for x in X:
...     for y in Y:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (2, 3), (2, 1), (3, 1)]
```

- It is equivalent to

```
>>> combs = \
... [(x, y) for x in X for y in Y if x != y]
```

List Comprehensions

- Other examples of list comprehensions:

```
>>> vec = [-4, -2, 0, 2, 4]
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> [(x, x**2) for x in range(5)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16)]
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Exercise 4-1: Transposition

- Consider the following example of a 3x4 matrix:

```
>>> [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

- Transpose rows and columns by list comprehension.

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

To generate a nested list, your comprehension should be similar to this form: `[ [ ... for ... ] for ... ]`

Try to fetch a column first. How to obtain `[1, 5, 9]`?

Sets

- A set is an unordered collection with no duplicate elements.
- Curly braces or the `set()` function can be used to create sets.

```
>>> basket = {'apple', 'orange', 'apple', 'pear',
'orange', 'banana'}
>>> print(basket) # the duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
```

- Check the membership by in keyword.

```
>>> 'orange' in basket
True
>>> 'crabgrass' in basket
False
```

Sets

- Mathematical operations like union (|), intersection (&), difference (−), and symmetric difference (^) are available.

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a      # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
```

- Similarly to list comprehensions, set comprehensions are also supported.

Dictionaries

- Dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys.
- A pair of braces creates an empty dictionary: `{}`.
- Placing a comma-separated list of `key:value` pairs within the braces adds initial `key:value` pairs to the dictionary.

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
```

Dictionaries

- If you want to iterate the dictionaries, use the `items()`, `keys()`, or `values()` function as your needed.
- Use the del keyword to delete some useless items:

```
>>> del tel['sape']
```

- The `dict()` constructor builds dictionaries directly from sequences of key-value pairs:

```
>>> dict([('sape', 4139), ('guido', 4127)])
```

- When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
>>> dict(sape=4139, guido=4127, jack=4098)
```

- Comprehensions can be used to create dictionaries.

Defining Functions

- The keyword `def` introduces a function definition.

  It must be followed by the function name and the parenthesized list of formal parameters.

  The statements that form the body of the function start at the next line, and must be indented.

- The first statement of the function body can optionally be a string literal, which is the documentation.
- The structure of a function is described as follows:

```
>>> def my_func(arguments):
...     """Documentation of my_func is optional"""
...     pass  # the body of this function
...
```

Exercise 5-1: Fibonacci Series

- Rewrite your "`fibonacci.py`", create a function that return a list of Fibonacci numbers.

```
$ cat fibonacci.py
#!/usr/bin/env python3
def fibnums(n):
    """Return n Fibonacci numbers"""
    fib = [ 0, 1 ]
    while len(fib) < n:
        pass  # Your previous solution
    return fib
print(fibnums(20))
```

More on Defining Functions

- The most useful form is to specify a default value for one or more arguments.

```
>>> def do_stuff(num, state='Empty', type='Blue'):
...     pass
```

- Functions can also be called using keyword arguments of the form kwarg=value.

```
do_stuff(1000)
do_stuff(num=1000)
do_stuff(num=1000000, state='General')
do_stuff(state='General', num=1000000)
do_stuff('a million', 'Ongoing', 'Green')
do_stuff('a thousand', type='Red')
```

More on Defining Functions

- The arbitrary number of arguments which could be received as a list (*args) or a dictionary (**args).

```
>>> def course(name, *students):
...     print('Students of', name)
...     for id, who in enumerate(students):
...         print(id, who)
...
>>> course("Deep Learning", \
... "James", "Lisa", "Thomas", "Maria")
Students of Deep Learning
0 James
1 Lisa
2 Thomas
3 Maria
```

Using Modules

- You may also want to use a handy function that you've written in several programs without copying its definition into each program.
- To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter.

  Such a file is called a module.

  Definitions from a module can be imported into other modules or into the main module.

Exercise 6-1: Fibonacci Module

- Your "`fibonacci.py`" is already a module.
- Launch the interpreter and try to import the Fibonacci module with following command:

```
>>> import fibonacci
```

- Then you are able to use the function defined in module.

```
>>> fib100 = fibonacci.fibnums(100)
>>> print(fib100)
```

- You can assign a function with a local name:

```
>>> fib = fibonacci.fibnums
```

- Or you may import the function from the module:

```
>>> from fibonacci import fibnums
>>> from fibonacci import fibnums as fib
```

More on Modules

- You may have noticed that the interpreter shows a list of 20 Fibonacci numbers when you import it.

```
$ cat fibonacci.py
#!/usr/bin/env python3
def fibnums(n):
    """Return n Fibonacci numbers"""
    pass
print(fibnums(20)) # here is the reason
```

- To make the file usable as a script as well as an importable module, place the statements under a "__main__ scope":

```
if __name__ == "__main__":
    print(fibnums(20))
```

More on Modules

- Python comes with a library of standard modules.

```
>>> import sys
```

- A collection of modules is usually packed in a package.

```
>>> import torch.nn.functional
>>> from torch.nn import functional
```

- The built-in function `dir()` is used to find out which names a module defines.
- You may use the built-in function `help()` to check the documentations.

```
>>> help(sys)
>>> help(list)
>>> help(list.append)
```

A First Look at Classes

- Classes provide a means of bundling data and functionality together.

  Creating a new class creates a new type of object.

  Class instances can also have methods (defined by its class) for modifying its state.

- The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

- In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed.

Class Instantiation

- The instantiation operation (“calling” a class object) creates an empty object.
- Many classes like to create objects with instances customized to a specific initial state.
- Therefore a class may define a special method named `__init__()`, like this:

```
class MyClass:
    def __init__(self):
        self.data = []
```

- In this example, a new instance can be obtained by:

```
x = MyClass()
```

Class Instantiation

- Of course, the `__init__()` method may have arguments for greater flexibility.

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Class Instantiation

- Sometimes it is useful to have a data type similar to the Pascal “record” or C “struct”, bundling together a few named data items.
- An empty class definition will do nicely:

```
class Employee:
    pass

john = Employee() # Create an empty record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Class and Instance Variables

- Consider the following implementation:

```
class Dog:
    kind = 'canine'
    def __init__(self, name):
        self.name = name
```

- Two kinds of variable:

  A class variable such as “kind” is shared by all instances.

  An instance variable such as “name” is unique to each instance.

- Shared data (class variables) can have possibly surprising effects with involving mutable objects such as lists and dictionaries.

  Do not modify them unless you know what you are doing.

Class Methods

- Often, the first argument of a method is called self.

  This is nothing more than a convention: the name self has absolutely no special meaning to Python.

- Methods may call other methods or instance variables by using method attributes of the self argument.

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Exercise 7-1: class Queue

- Create a class Queue, implement `pop()`, `push()`, `peek()`, `size()` with list.

  Your implementation should use a list as an instance variable for storing the items in the queue.

- The structure of a class:

```
class NameOfNewClass:
    class_variable = 'foo'
    def __init__(self, bar='bar'):
        self.instance_variable = bar
    def some_method(self, arg):
        pass
```

- Design the constructor with default argument values.

  Accept both `Queue()` and `Queue([0, 1, 2])`.

Inheritance

- Of course, a language feature would not be worthy of the name “class” without supporting inheritance.
- The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

- The class BaseClassName must be defined in a scope containing the derived class definition.
- Two built-in functions that work with inheritance: `isinstance()` and `issubclass()`.

Overriding Methods

- Derived classes may override methods of their base classes.

  For C++ programmers: all methods are effectively virtual.

- An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name.

- There is a simple way to call the base class method directly:

```
BaseClassName.methodname(self, arguments)
```

Exercise 7-2: class Stack(Queue)

- Create a class Stack which inherit the class Queue you previously defined and overriding some methods for Stack.

  Do not modify the list object in the Queue directly.
  Try to access Queue's methods in class Stack.

- Try to create another class method whose name is already exists in the base class, but change the arguments of the method.

  In Queue class: `def pop(self)`
  In Stack class: `def pop(self, num_pop=1)`

- Which method will be invoked when you call `Stack.pop()`?

Virtual Environments

- Applications will often use packages and modules that don't come as part of the standard library.
- It may not be possible for one Python installation to meet the requirements of every application.

  If application A needs version 1.0 of a particular module but application B needs version 2.0, then the requirements are in conflict.
- The solution is to create a virtual environment, a self-contained directory tree that contains a Python installation for a particular version, plus a number of additional packages.

Virtual Environments

- To create a virtual environment, decide upon a directory where you want to place it, and run the venv module as a script with the directory path:

```
$ python3 -m venv tutorial-env
```

- Once you've created a virtual environment, you may activate it.

```
$ source tutorial-env/bin/activate
(tutorial-env) $ python # run the interpreter or scripts
Python 3.6.2 (default, Jul 22 2017, 21:19:22)
  ...
```

- To leave the virtual environment, use deactivate:

```
(tutorial-env) $ deactivate
```

Exercise 8-1: Using the Virtual Env.

- Create two virtual environments, one for Python 3 and another for Python 2:

```
$ python3 -m venv ~/venv-3
$ python2 -m venv ~/venv-2
```

- Activate them and run your scripts.

```
$ source ~/venv-3/bin/activate
(venv-3) $ # run the interpreter or scripts
(venv-3) $ deactivate
$ source ~/venv-2/bin/activate
(venv-2) $ # run the interpreter or scripts
(venv-2) $ deactivate
```

Managing Packages with pip

- You can install, upgrade, and remove packages using a program called pip.

  By default pip will install packages from the Python Package Index, <https://pypi.python.org/pypi>.

- Browse the Python Package Index by going to it in your web browser, or use pip's limited search feature:

```
$ pip search numpy
```

- Install the latest version of a package:

```
$ pip install numpy
```

- Display all of the packages installed:

```
$ pip list
```

Exercise 8-2: pip and NumPy

- Activate your virtual environment, check the installed packages, and install NumPy by pip.

```
$ source ~/venv-3/bin/activate
(venv-3) $ pip list
... # numpy should not be here
(venv-3) $ pip install numpy
...
Installing collected packages: numpy
Successfully installed numpy-1.13.3
(venv-3) $ pip list
... # numpy should be here now
```

- Launch the Python interpreter, import numpy.

```
(venv-3) $ python
>>> import numpy as np
```

Case Study: DDPG

- You are able to “read” a real-world Python program of deep learning now.
- Case study — DDPG:

```
$ git clone https://github.com/pemami4911/deep-rl.git
$ vim deep-rl/ddpg/replay_buffer.py
$ vim deep-rl/ddpg/ddpg.py
```

- Note that you do not need to understand what the script actually doing.

  Focus on the grammar and structure of the program.

  Try to find the documentation of any pieces that you are interested.

Exercise 9-1: Matrix Multiplication

- Write a Python script to do matrix multiplication.

  You just installed NumPy in the virtual environment, which is a module for handling multidimensional array.

- Your script should ask the size and the entries of two matrices.

  If the two matrices are incompatible, or the entries and the size is mismatched, display a message and exit.

- Once the matrices are ready, produce the product.

  Hints: `numpy`, `numpy.array()`, `numpy.reshape()`, `numpy.ndarray`, `numpy.ndarray.dot()`, `input()`, `len()`, `str.split()`, list comprehensions.

What Now?

- Learning this tutorial has probably reinforced your interest in using Python — you should be eager to apply Python to solving your real-world problems.
- Note that this tutorial is modified from the official tutorial, and should not be used for other purposes.
- Where should you go to learn more?

  The Python Tutorial (official)

  The documentation for Python

  PyTorch Tutorials