PYTHON INTRODUCTION

Preliminary

- Mainly target python 2.7
- Why?
  - Compatibility: a few of packages/libraries still have no support for python 3
  - Performance: https://speakerdeck.com/pyconslides/python-3-dot-3-trust-me-its-better-than-python-2-dot-7-by-dr-brett-cannon
    - The performance of python 2.7 is still better than python 3.3 in my opinion
- The EOL date for python 2.7 is 2020, so I'm not worry about it so far :P

Preliminary

- Target audience
  - Experienced programmer who is familiar with an imperative, structured or object-oriented programming language

Let's Start

- One good way to thoroughly understand a programming language is "studying the language reference/specification"
  - Yes, I mean it.
- Python 2.x language reference: https://docs.python.org/2/reference/index.html

Fundamentals of Programming Language

- Data model
- Control flow / program structure
- Primitive support for a programming paradigm

Basic Syntax

- https://docs.python.org/2/reference/lexical_analysis.html
- Summary
  - No “;” at the end of line, use lines (line break)
  - Join lines: “\” or in “()”, “[]” or “{}”
  - Use indentation & blank lines to compose/separate code blocks
  - String: prefix “r”: raw, “u”: Unicode, “b”: bytes
  - Integer: 32 (32 bits singed integer), 64L/64l (64 bits signed integer)
  - Float: equal to “double” in C/C++ or Java

DATA MODEL

Types & Objects

- Each object has an identifier, a type and a value
- An attribute is a value associated with an object
  - Ex. object.attr_val
- A method is a function that performs some sort of operation on an object when the method is invoked as a function
  - Ex. object.add(x)
- Get type: type(x)
- Check type: isinstance(x, list)

Types & Objects

- None: type(None), None -> null
- Numbers
  - int(): 32-bit integer
  - long(): 64-bit integer
  - float(): “double” floating point
  - bool(): “True” or “False”
  - complex(): complex number

Types & Objects

- Sequences
  - Immutable
    - str(): string
    - unicode(): Unicode
    - tuple(): looks like a immutable list, but it’s not. (talk about it later)
  - Mutable
    - list(): just list
    - bytearray(): like char * in C/C++
- Mapping
  - dict(): dictionary, like a hash table
- Set
  - set(): mutable set
  - frozenset(): immutable set

Types & Objects

- Callable types
  - User-defined functions: first class object
    - It’s very powerful
  - User-defined methods
  - Generator functions
  - …

    (We will talk about this later)

Sequences

- Operations & methods
  - Common
    - https://docs.python.org/2/library/stdtypes.html#typesseq
  - Mutable sequence type
    - https://docs.python.org/2/library/stdtypes.html#typesseq-mutable
- Use an index to access the element in a sequence type: s[i]
- Slice
  - s[start:stop]: a slice started
  - s[start:stop:step]: a extended slice

Sequences: String

- Operations & methods
  - https://docs.python.org/2/library/stdtypes.html#string-methods
- The tricky part of string is the string formatting
- First, the simple one
  - https://docs.python.org/2/library/stdtypes.html#string-formatting-operations
  - Use “%” operator
  - '%(language)s has %(number)03d quote types.' % {"language": "Python", "number": 2}
    - Argument is a dictionary
  - '%s has %03d quote types.' % ("Python", 2)
    - Argument is a tuple
- .format() method
  - '{:s} has {:03d} quote types.'.format("Python", 2)
- https://pyformat.info/
- Format method is more powerful.

Sequences: Tuple & List

- Tuple: (elem1, elem2, elem3)
  - Immutable
  - Usually contain a heterogeneous sequence of elements
  - Accessed via unpacking or indexing
  - Packing: t = ‘one’, 2, 3.0
  - Unpacking: s, i, f = t
- List: [elem1, elem2, elem3]
  - Mutable
  - Usually contain a homogeneous sequence of elements
  - Accessed by iterating over the list
- NamedTuple
  - Accessed by attribute lookup
  - https://docs.python.org/2/library/collections.html#collections.namedtuple

Mapping: Dictionary

- Operations & methods
  - https://docs.python.org/2/library/stdtypes.html#mapping-types-dict
- Use a “key” to access an element in a dictionary : d[key]
- Key must be a “hashable” value:
  - https://docs.python.org/2/glossary.html#term-hashable
  - “An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a __hash__() method), and can be compared to other objects (it needs an __eq__() or __cmp__() method). Hashable objects which compare equal must have the same hash value.”
- E.g. d = {‘one’: 1, ‘two’: 2, ‘three’: 3}

Set

- Operation & methods
  - https://docs.python.org/2/library/stdtypes.html#set-types-set-frozenset
- A set object is an unordered collection of distinct ”hashable” objects (use the same definition in the last page)

CONTROL FLOW / PROGRAM STRUCTURE

Expressions

- Arithmetic & bitwise operations
  - x + y, x - y, x * y, x / y, x % y, -x, +x
  - x ** y: power of x
  - x // y: floor division, like integer division in C
  - x << y, x >> y, x & y, x | y, x ^ y (xor), ~x
- Comparison
  - x < y, x > y, x == y, x != y, x >= y, x <= y
- Boolean operations
  - x or y
  - x and y
  - not x
  - Short-circuit evaluation, just like C/C++, Java
- Operator precedence
  - https://docs.python.org/2/reference/expressions.html#operator-precedence

Expressions

- Conditional expression
  - max = x if x > y else y
  - Similar “max = x > y ? X : y;” in C/C++
- Will talk about 3 important expressions in later sections
  - List comprehension
  - Lambda expression
  - Yield expression

Statements

- Assignment statement
  - x = y, x += y, x-= y, x *= y, x /= y, x //= y, x %= y, x **=y, x <<= y, x >>=y, x &= y, x |= y, x ^= y
- Assert statement

```
assert expression
# equal to
if __debug__:
    if not expression:
        raise AssertionError
```

- Pass statement
  - NOP

```
def f(arg):
    pass
```

Compound Statements

- if-then-else

```
if expression:
    statements
elif expression:
    statements
else:
    statements
```

- while
  - “continue” statement
  - “break” statement

```
while expression:
    statements
```

Compound Statements

- for loop
  - “target_list”: “x, y, z”, separated by “,”
  - “expression_list” should yield a iterable object
  - else: executed when there is no break occurred

```
for target_list in expression_list:
    statements

for x in s: # s is a object of sequence type
    print x

for key, value in d.iteritems(): # d is a dictioary
    print 'key=%s,value=%s' % (key, value)

for x in s:
    if x == 10:
        break
else:
    print '10 is NOT FOUND'
```

Compound Statements

- try: exception handling

```
try:
    statements
except ValueError as e:
    statements
except TypeError, e:
    statements
else:
    # not in preceding exception types
    statements
finally:
    # always executed
    statemetns
```

Compound Statements

- with statement
  - Execute inside a runtime context
  - Need a object served as context manager, which has two methods:
    - object.__enter__(self)
    - object.__exit__(self, exc_type, exc_value, trackback)
    - https://docs.python.org/2/reference/datamodel.html#context-managers

with statement

```
with expression as ctx:
    statemetns

with A() as a, B() as b:
    statements
# equal to
with A() as a:
    with B() as b:
        statements

with open(filename) as f:
    f.write(string)
    ...

lock = threading.Lock()
with lock:
    # critical section
    statements
    # end of critical section
```

SUPPORT FOR PROGRAMMING PARADIGM

FUNCTION & FUNCTIONAL PROGRAMMING

Function

- Parameters
  - Pass by value + pass by reference
  - Default parameter values: optional parameters
    - Parameter which has a default value is a optional parameter
    - Optional parameter must be defined after those without default values (similar to C++)
  - Invocation with parameters
    - Position arguments
    - Keyword arguments
    - Keyword arguments must be located after position arguments

Function

```
def func(args):
    statements

def func(arg1, arg2='default'): # parameter with default value
    statements

# invoke with mixed position argument & keyword argument
func(arg1='arg1', 'arg2')
func('arg1', arg2='arg2')

def func(arg1, *args, **kargs):
    # *args & **kargs are optional

    # position argument, args is a tuple
    for x in args:
        pass

    # keyword argument, kargs is a dictionary
    for k, v in kargs.iteritems():
        pass

func('arg1')
# args => (), kargs => {}

func('arg1', 'arg2')
# args => ('arg2'), kargs => {}

func('arg1', arg='arg2')
# args => (), kargs => {'arg': 'arg2'}

func('arg1', 'arg2', arg='arg3')
# args => ('arg2'), kargs => {'arg': 'arg3'}
```

Function

- Expand list or dictionary as position or keyword arguments
  - ‘*’ for position arguments
  - ‘** for keyword arguemtns

```
def func(arg1, arg2, arg3):
    pass

args = ['arg1', 'arg2', 'arg3']
func(*args) # expand postional arguments

args = {'arg1': 'arg1', 'arg2': 'arg2', 'arg3': 'arg3'}
func(**args) # expand keyword arguments
```

Function

- Variable scope
  - Variables defined in a function are local variables
  - Local variable will hide the outer variable with the same name
  - “global variable”: declare a global variable in a function

```
g_var = 1
def func():
    g_var = 2
    print g_var # g_var = 2

func()
print g_var # g_var = 1

g_var = 1
def func():
    global g_var
    g_var = 2

func()
print g_var # g_var = 2
```

Function

- Nested function
  - Be careful of variable scope: only local & global in Python 2.x (Python 3 has “nonlocal” variable)

```
def func():
    def func_1():
        statements

    func_1()
    statements
```

- return statement
  - “return expression_list”
  - Return an expression_list means it can return a tuple, list, dict or etc.

```
def func():
    return 'string', 1, 1.2 # return a tuple

s, i, f = func() # unpack tuple
```

Functional Programming

- Directly jump to the conclusion:
  - Functional programming is the future, period.
- Functions are first-class objects in Python (!!! It's very important !!!)
  - Can be passed as arguments
  - Can be placed in data structures
  - Can be returned as results
- Closures
  - The value of a variable in a function is bound to the context of when constructing the function object

Closures

```
val = 20
def add_header(func):
    print '#####'
    return func()

val = 10
def gen_string():
    print 'value = %d' % (val)

add_header(gen_string) # val is bound to 10

def gen_multiply():
    base = 2
    def multiply(val):
        return base * val
    return multiply

f = get_multiply(2)
base = 10
f(3) # => 6, the base is bound to 2

def gen_multiply(base):
    def multiply(val):
        return base * val
    return multiply

f = get_multiply(2)
f(3) # => 6
```

Function Decorator

- Decorator is a wrapper of function, it’s a syntax sugar

```
@foo
def func():
    pass

# equal to
def func():
    pass

# function foo() should be defined somewhere
func = foo(func)

# cascading
@foo
@bar
def func():
    pass

# equal to
func = foo(bar(func))
```

Generator & yield statement/expression

- yield [expression_list]
- Create a generator function
  - Call a generator function, it returns a generator (a kind of iterator)
  - Use generator.next(), StopIteration exception to control execution flow
  - generator.next() -> execute function until “yield” expression -> return expression_list -> loop back until return or has a GeneratorExit exception
  - Caller will get StopIteration (callee return normally) or other exception
- Generator methods:
  - next()
  - send()
  - throw(): raise an exception
  - close(): stop iteration

Generator

```
def countdown(val):
    while val > 0:
        yield val
        val -= 1

g = countdown(10)
g.next() # 10
g.next() # 9
...
g.next() # 1
g.next()
Traceback (most recent call last):
  g.next()
StopIteration

for n in countdown(10):
    print n # 10, 9, ... , 1
```

Coroutine & yield expression

- What's coroutine : https://en.wikipedia.org/wiki/Coroutine
- Why I should know about it : it's very powerful in some kinds of applications
  - Concurrency
  - Dataflow programming
  - Golang's goroutine is quite similar to coroutine
- It can assign/get value by a yield expression

Coroutine

```
def receiver():
    print 'Ready to receive'
    try:
        while True:
            n = (yield)
            print 'Got %d' % (n)
    except GeneratorExit:
        print 'Receiver done'

r = receiver()
r.next()
# Ready to receive
r.send(1)
# Got 1
r.send(2)
# Got 2
r.close()
# Receiver done

def splitter():
    print 'Ready to split'
    try:
        result = ['ready']
        while True:
            line = (yield result)
            result = line.split()
    except GeneratorExit:
        print 'Splitter done'

s = splitter()
s.next()
# Ready to split, ['ready']
s.send('col1 col2 col3')
# [col1, col2, col3]
s.close()
# Splitter done
```

List Comprehension

- Construct a list in one expression, math-like syntax

```
[expression for item1 in iter1 if expr_cond1
            for item2 in iter2 if expr_cond2
            ...
            for itemN in iterN if expr_condN]
# if expr_condN is optional

s = [(x, y) for x in 'abc' for y in xrange(0,5) if y % 2 == 0]
# [('a', 0), ('a', 2), ('a', 4),
#  ('b', 0), ('b', 2), ('b', 4),
#  ('c', 0), ('c', 2), ('c', 4)]
```

- There are also set comprehension, dictionary comprehension
  - http://treyhunner.com/2015/12/python-list-comprehensions-now-in-color/

Generator Expression

- Construct a generator in one expression

```
(expression for item1 in iter1 if expr_cond1
              for item2 in iter2 if expr_cond2
              ...
              for itemN in iterN if expr_condN)
g = ((x, y) for x in 'abc' for y in xrange(0,5) if y % 2 == 0)
g.next() # ('a', 0)
g.next() # ('a', 2)
...
```

Lambda Expression

- Similar to anonymous function in JavaScript, but it's a expression, not a statement
- lambda [parameter_list] : expression

```
f = lambda x, y: x ** y
f(2,10) # 1024
```

- Gossip: why called lambda?
  - Lambda calculus
    - https://en.wikipedia.org/wiki/Lambda_calculus

Summary

- Functional programming is the future, period.

OBJECT-ORIENTED PROGRAMMING

Object-Oriented Programming

- (Err… I don't like OO, so … whatever)
- Class definition

```
class Foo(object):
    num_element = 0 # class variable, shared by all instances

    # constructor
    def __init__(self, name, address):
        self.name = name
        self.addr = address
        Foo.num_element += 1

    # destructor
    def __del__(self):
        Foo.num_element -= 1

    def bar(self, val):
        pass

    def scope(self, val):
        self.bar(val)
        Foo.scope(self, val)

obj = Foo('foo', 'bar')
```

Class

- Each class has one class object
- Instance: “self”
  - Constructor: __init__(self, …)
  - Destructor: __del__(self)
  - Memory management: __new__(cls, *args, **kargs)
- Call method
  - self.method()
  - Class.method(self)

Inheritance

- Support multiple inheritances (Damn it…)

```
class Foo(Base1, Base2, Base3):
    def __init__(self, name):
        self.name = name
    def bar(self, val):
        # Base1 has bar() or
        # one of the parents of Base1 has bar()
        Base1.bar(self, val)
        super(Foo,self).bar(val)
```

- super() to find the right base class of a method
  - How? C3 linearization algorithm (WTH…)
  - https://en.wikipedia.org/wiki/C3_linearization
  - Checkout by DerivedClass.__mro__

Dynamic binding and Duck Typing

- “If it looks like, quacks like, and walks like a duck, then it’s a duck.”

```
def execute(obj, val):
    obj.run()
    obj.stop()

class Foo(object):
    def run(self, val):
        pass
    def stop(self):
        pass

class Bar(object):
    def run(self, val):
        pass
    def stop(self):
        pass

obj1 = Foo()
obj2 = Bar()
execute(obj1, 'Man')
execute(obj2, 0xBEEF)
```

Private Variables

- Attribute prefixed by at least 2 underscores (variable or method name)

```
class Bar(object):
    def run(self):
        self.__do_run()

    def do_run(self):
        print 'Bar run'

    __do_run = do_run

class Foo(Bar):
    def do_run(self):
        print 'Foo run'

obj = Foo()
obj.run() # Bar run
```

Operator Overloading

- https://docs.python.org/2/library/operator.html

```
class Foo(object):
    def __add__(self, b):
        pass
    def __del__(self, b):
        pass
    ...

x = Foo()
x + 4 # => Foo.__add__(self, 4)
```

Special Methods

- Static method: @staticmethod
  - Use class name as a namespace
- Class method: @classmethod
  - Operate on class (class object), not object instance

```
class Foo(object):
    num_element = 0
    @staticmethod
    def add(x, y):
        return x + y
    @classmethod
    def add_element(cls, val):
        cls.num_element += val

Foo.add(1, 2) # 3
Foo.add_element(1) # 0 -> 1
```

Property

- @property: getter
- @prop_name.setter: setter of prop_name
- @prop_name.deleter: deleter of prop_name

```
class Foo(object):
    def __init__(self):
        self._x = 0

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

Class Decorator

- Similar to function decorator

```
all_cls = []
def record(cls):
    all_cls.append(cls.__cid__)
    return cls

@record
class Foo(object):
    __cid__ = 'Foo'
    ...

# equal to

class Foo(object):
    __cid__ = 'Foo'
    ...
Foo = record(Foo)
```

OTHER TOPICS

Module

- “import” module
- “from” module “import” symbol

```
import os
os.getcwd()

from os import getcwd
getcwd()

import subprocess as sp
sp.call()

from subprocess import Popen as pp
pipe = pp('pwd')

# import muliple modules or symbols in one line
import os, subprocess as sp
from subprocess import Popen as pp, STDOUT
```

Package

- Package initialization: __init__.py in each directory under package path

```
### Directory tree
lib/
    __init__.py
    one/
        __init__.py
        run.py

###
lib/one/run.py:

class Runner():
    pass
###

import lib.one.run
lib.one.run.Runner()

from lib.one import run
run.Runner()

from lib.one.run import Runner
Runner()
```

Unit Testing

- unittest module
  - https://docs.python.org/2/library/unittest.html
- Steps:
  - Create subclasses of unittest.TestCase
  - Create test cases as methods which method name is started with “test*”
  - Call unittest.main()
- More practices for unittest in later Django study, Django also use unittest as unit test framework

Coverage

- Coverage tool is quite useful for improving the quality of test cases.
- Coverage.py
  - https://coverage.readthedocs.io/en/coverage-4.2/
- Easy to use through CLI command or API
- Sample report
  - http://nedbatchelder.com/files/sample_coverage_html/index.html