Python Course

June 14, 2018

1 Python Course

```
In [2]: import os
        os.getcwd()

Out[2]: '/Users/fanggj/gitrepo/LC102-Python/python_course/lecture00'
```

1.1 Course goals

- The audience could implement a specific feature in a elegant and clean way.
- The audience could use the right tools in order to make life easier and happier.

1.2 Prerequisite

- Python syntax.

2 Reading Recommendations

2.1 Overview

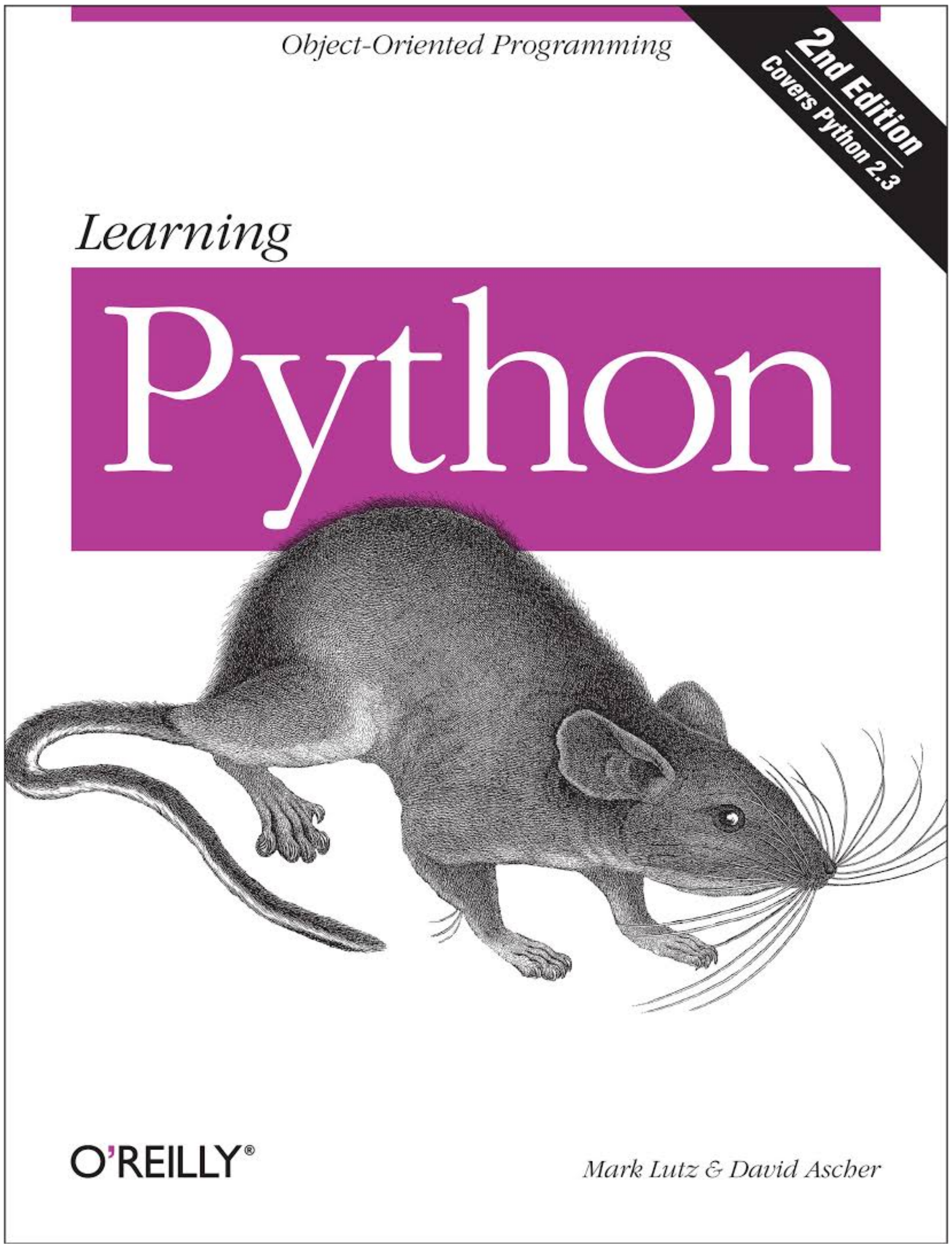
In Python, variables are simply names that refer to objects. Variables do not need to be declared before they are assigned and they can even change type in the middle of a program. Like other dynamic languages, all type-checking is performed at run-time by an interpreter instead of during a separate compilation step.

2.2 Lexical analysis

Although it's easy to start writing Python code with merely basic syntax knowledge, I'd like to start our course from the lexical analysis of Python to make sure we didn't miss anything important.

2.2.1 Encoding declarations

```
# -*- coding: <encoding-name> -*-
```



"learning python"

If a comment in the first or second line of the Python script matches the regular expression `coding[=:]\s*([-\w.]+)`, this comment is processed as an encoding declaration; the first group of this expression names the encoding of the source code file. The encoding declaration must appear on a line of its own. If it is the second line, the first line must also be a comment-only line.

UTF-8 is the default source encoding for Python 3, just as ASCII was the default for Python 2 (starting with 2.5).

The default encoding behavior is defined in the function `decoding_fgets` which is defined in the Parser/tokenizer.c file. This function was added in the commit below:

- Patch #534304: Implement phase 1 of PEP 263. https://github.com/python/cpython/commit/00f1e3f5a54a 30b8266a4285de981f8b1b82a8cc6231

Before Python 2.5, there is only `fgets` available to read lines from source file.

- Mass checkin of universal newline support. https://github.com/python/cpython/commit/7b8c7546ebc1f 30b8266a4285de981f8b1b82a8cc6231

```
char *
Py_UniversalNewlineFgets(char *buf, int n, FILE *stream, PyObject *fobj)
{
    return fgets(buf, n, stream);
}
```

2.2.2 Indentation

In Python 2, one table is an equivalence of eight spaces. In Python 3, can not use a mixture of tabs and spaces in indentation.

Make tabs always 8 spaces wide – it's more portable. https://github.com/python/cpython/commit/4fe872988b3dd9edf004160c44076df839f14516#diff-30b8266a4285de981f8b1b82a8cc6231

```
In [4]: for i in range(3):
                print(i) # 8 spaces this line.
                print(i) # 1 tab this line.

0
0
1
1
2
2
```

Never ever use a mixture of tabs and spaces.

2.2.3 Reserved classes of identifiers

`_*`

Not imported by `from module import *`. The special identifier _ is used in the interactive interpreter to store the result of the last evaluation; it is stored in the `__builtin__` module. When not in interactive mode, _ has no special meaning and is not defined.

`__*__`

System-defined names. These names are defined by the interpreter and its implementation (including the standard library). Current system names are discussed in the Special method names section and elsewhere. More will likely be defined in future versions of Python. Any use of `__*__` names, in any context, that does not follow explicitly documented use, is subject to breakage without warning.

`__*`

Class-private names. Names in this category, when used within the context of a class definition, are re-written to use a mangled form to help avoid name clashes between "private" attributes of base and derived classes.

2.3 Data Types

In this section, I will introduce some basic Python types which come from the `types` module.

```
In [1]: import types
        dir(types)
```

```
Out[1]: ['BooleanType',
         'BufferType',
         'BuiltinFunctionType',
         'BuiltinMethodType',
         'ClassType',
         'CodeType',
         'ComplexType',
         'DictProxyType',
         'DictType',
         'DictionaryType',
         'EllipsisType',
         'FileType',
         'FloatType',
         'FrameType',
         'FunctionType',
         'GeneratorType',
         'GetSetDescriptorType',
         'InstanceType',
         'IntType',
         'LambdaType',
         'ListType',
         'LongType',
         'MemberDescriptorType',
         'MethodType',
         'ModuleType',
         'NoneType',
         'NotImplementedType',
         'ObjectType',
         'SliceType',
         'StringType',
         'StringTypes',
         'TracebackType',
         'TupleType',
         'TypeType',
         'UnboundMethodType',
         'UnicodeType',
         'XRangeType',
         '__all__',
         '__builtins__',
         '__doc__',
         '__file__',
         '__name__',
         '__package__']
```

In order to discuss them in a proper way, we will classify them into several groups:

- Numeric Types
- Sequence Types
- Set Types and Mapping Types
- Other types

2.3.1 Numeric Types

Built-in numeric types There are some numeric types which can be defined as a literal. Please note that `bool` type is a numeric type.

```
In [17]: # built-in numeric types
         {
             type(True): True,
             type(1): 1,
             type(1.0): 1.0,
             type(1+1j): 1+1j,
             type(1L): 1L,
         }

Out[17]: {bool: True, complex: (1+1j), float: 1.0, int: 1, long: 1L}

In [19]: issubclass(bool, int)

Out[19]: True
```

Other numeric types There are some other numeric types in Python Standard Library, for example, Decimal and Fraction.

Besides the regular use cases of the numbers, there are some singular cases to which we need to pay attention.

```
In [15]: # other numeric types
         from decimal import Decimal
         from fractions import Fraction

         Fraction(2, 3), Decimal('1.000000001')
```

```
Out[15]: (Fraction(2, 3), Decimal('1.000000001'))
```

Number tricks

boolean There are three types of integers: - Integers (int or long) - Booleans

PEP 285 – Adding a bool type

This PEP proposes the introduction of a new built-in type, bool, with two constants, False and True.

The bool type would be a straightforward subtype (in C) of the int type, and the values False and True would behave like 0 and 1 in most respects (for example, False==0 and True==1 would be true) except repr() and str().

All built-in operations that conceptually return a Boolean result will be changed to return False or True instead of 0 or 1; for example, comparisons, the "not" operator, and predicates like isinstance().

```
In [26]: issubclass(bool, int), str(True), repr(True)
```

```
Out[26]: (True, 'True', 'True')
```

```
In [19]: True == 1, True & 2, True | 2, True << 2, True >> 1, ~True, ~False, -True
```

```
Out[19]: (True, 0, 3, 4, 0, -2, -1, -1)
```

```
In [14]: True + 2, True - True, True * 2, 2 / True, True ** True
```

```
Out[14]: (3, 0, 2, 2, 1)
```

```
In [20]: import math
         math.cos(True)
```

```
Out[20]: 0.5403023058681398
```

Because True is equal to 1 and False is equal to 0, there is a tricky scenario when you use bool as the key of a dictionary.

```
In [22]: {True:'True', 1:'One'}
```

```
Out[22]: {True: 'One'}
```

In Python, bools are not 'textbook' bools which do not support arithmetic operations. Please keep that in mind.

float Almost all platforms map Python floats to IEEE-754 “double precision”. In that way, the represent of the floats will encounter the precision issue.

https://docs.python.org/2/tutorial/floatingpoint.html

```
In [15]: from fractions import Fraction
         print(0.1 + 0.2, Fraction(0.1 + 0.2))
```

```
(0.30000000000000004, Fraction(1351079888211149, 4503599627370496))
```

By using Decimal, which is a fixed point, the precision issue can be solved.

```
In [1]: from decimal import Decimal
        from fractions import Fraction
        # Don't use float number to construct the Decimal.
        Decimal("0.1") + Decimal("0.2"), Fraction(Decimal("0.1") + Decimal("0.2"))
```

```
Out[1]: (Decimal('0.3'), Fraction(3, 10))
```

Although Decimal is a handy class to use, but it seems that in some cases Decimal cannot be used directly.

```
In [18]: import numbers
         from decimal import Decimal

         issubclass(Decimal, numbers.Real), issubclass(Decimal, numbers.Number)
```

```
Out[18]: (False, True)
```

```
In [17]: import json
         from decimal import Decimal

         json.dumps({1: Decimal("1")})
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-17-9ea8cead75a8> in <module>()
          2 from decimal import Decimal
          3
    ----> 4 json.dumps({1: Decimal("1")})


        /usr/local/Cellar/python@2/2.7.15/Frameworks/Python.framework/Versions/2.7/lib/python2
        242         cls is None and indent is None and separators is None and
        243         encoding == 'utf-8' and default is None and not sort_keys and not kw):
    --> 244         return _default_encoder.encode(obj)
```

```
        245     if cls is None:
        246         cls = JSONEncoder


        /usr/local/Cellar/python@2/2.7.15/Frameworks/Python.framework/Versions/2.7/lib/python2
        205         # exceptions aren't as detailed.  The list call should be roughly
        206         # equivalent to the PySequence_Fast that ''.join() would do.
    --> 207         chunks = self.iterencode(o, _one_shot=True)
        208         if not isinstance(chunks, (list, tuple)):
        209             chunks = list(chunks)


        /usr/local/Cellar/python@2/2.7.15/Frameworks/Python.framework/Versions/2.7/lib/python2
        268                 self.key_separator, self.item_separator, self.sort_keys,
        269                 self.skipkeys, _one_shot)
    --> 270         return _iterencode(o, 0)
        271
        272 def _make_iterencode(markers, _default, _encoder, _indent, _floatstr,


        /usr/local/Cellar/python@2/2.7.15/Frameworks/Python.framework/Versions/2.7/lib/python2
        182
        183         """
    --> 184         raise TypeError(repr(o) + " is not JSON serializable")
        185
        186     def encode(self, o):


        TypeError: Decimal('1') is not JSON serializable
```

As a part of IEEE-754, there are nan and inf. Although nan may be not very useful in our real world, but the inf is still a useful concept.

```
In [25]: inf = float('inf')
         inf + inf, inf - inf, inf * inf, inf / inf, -inf

Out[25]: (inf, nan, inf, nan, -inf)
```

2.3.2 Sequence Types

Mutable sequences vs. Immutable sequences An object of an immutable sequence type cannot change once it is created.

Mutable sequences can be changed after they are created.

```
In [5]: import collections

        print "name\tmutable?"
        for t in [str, list, tuple]:
            print "%s\t%s" % (t.__name__, issubclass(t, collections.MutableSequence))
```

```
name        mutable?
str        False
list        True
tuple        False
```

```
In [2]: # This is different from C/C++
        s = 'Hello'
        s[0] = 'h'
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-2-b6c70510af66> in <module>()
          1 # This is different from C/C++
          2 s = 'Hello'
    ----> 3 s[0] = 'h'


        TypeError: 'str' object does not support item assignment
```

If the object contains references to other objects, these other objects may be mutable and may be changed; however, the collection of objects directly referenced by an immutable object cannot change.

```
In [8]: a = (1, [], 'Hello')
        a[1].append(2) # The refered object can be changed.
        print(a)
        a[1] = [] # The reference itself in the immutable sequence can not be changed.
```

```
(1, [2], 'Hello')
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-8-6c0266194b77> in <module>()
          2 a[1].append(2) # The refered object can be changed.
          3 print(a)
    ----> 4 a[1] = [] # The reference itself in the immutable sequence can not be changed.


        TypeError: 'tuple' object does not support item assignment
```

List Comprehensions

```
In [24]: # Variable leaking in Python 2
         l = [x for x in range(10)]
         print(l)
         print(x)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
9
```

Slicing `a[i:j:k]` selects all items of a with index x where x = i + n*k, n >= 0 and i <= x < j.

To evaluate the expression seq[start:stop:step], Python calls seq.`__getitem__`(slice(start, stop, step)).

Slice objects are used to represent slices when extended slice syntax is used. This is a slice using two colons, or multiple slices or ellipses separated by commas, e.g., a[i:j:step], a[i:j, k:l], or a[. . . , i:j]. They are also created by the built-in `slice()` function.

```
In [36]: l = range(10)
         s = slice(1, 100, 2)
         print(l[s])
         print(s.indices(len(l)))
```

```
[1, 3, 5, 7, 9]
(1, 10, 2)
```

```
In [44]: class L(object):
             def __getitem__(self, index):
                 print(index)

         l = L()
         l[1:2]
         l[1:2, 3:4]
         l[1, ..., -1]
```

```
slice(1, 2, None)
(slice(1, 2, None), slice(3, 4, None))
(1, Ellipsis, -1)
```

Sequence tricks Some operations on the sequences have very tricky. There are some misusing cases below.

```
In [55]: # create a two dimensions matrix
         m = [[0] * 2] * 2
         m[0][0] = 1
         print(m)
```

```
[[1, 0], [1, 0]]
```

The reference has been copied.

When performing an augmented assignment, if the object is mutable, it will be changed in place.

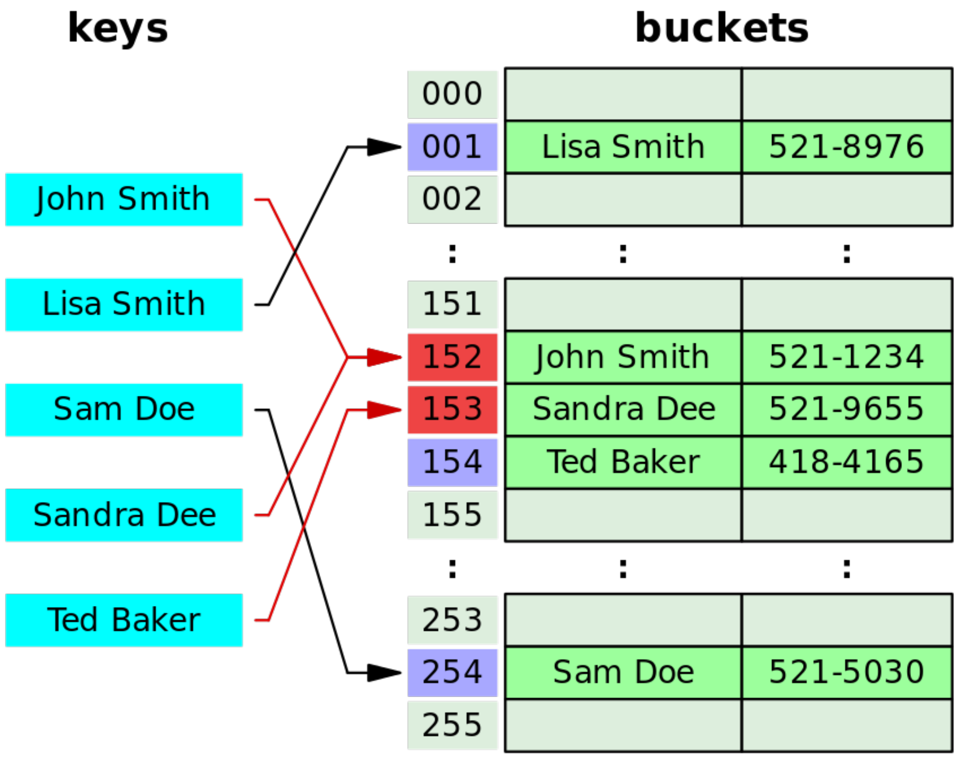
```
In [82]: # augmented assignment
         m = (1, 2, [3, 4])
         m[2].append(5)
         print(m)
         try:
             m[2] += [6]
         except:
             import traceback
             print(traceback.format_exc())
         print(m)
```

```
(1, 2, [3, 4, 5])
Traceback (most recent call last):
  File "<ipython-input-82-1feba7fbae78>", line 6, in <module>
    m[2] += [6]
TypeError: 'tuple' object does not support item assignment

(1, 2, [3, 4, 5, 6])
```

Let's see how to fix them.

```
In [75]: def create_matrix(dimensions):
             """
             create the matrix recursively.

             Question: how many times the create_matrix function will be called if I want crea
             """
             if not dimensions:
                 return [] # empty matrix
             if len(dimensions) == 1:
                 return [0] * dimensions[0] # new list created and returned.
             return [create_matrix(dimensions[1:]) for _ in range(dimensions[0])] # repeatedly

         m = create_matrix((2, 2))
         m[0][0] = 1
         print(m)
```

```
[[1, 0], [0, 0]]
```



open addressing with linear probing

```
In [83]: # Never place any mutable object in a immutable container.
         # Instead of modifying the immutable object, please re-create a new one.
         m = (1, 2, (3, 4))
         m = m[:2] + (m[2] + (5,),)
         print(m)
         m = m[:2] + (m[2] + (6,),)
         print(m)
```

```
(1, 2, (3, 4, 5))
(1, 2, (3, 4, 5, 6))
```

2.3.3 Set types and Mappings

Set types and mappings are implemented using hash tables.
Hash Table Example: Hash collision resolved by open addressing with linear probing
Collision resolution in Python: Open addressing with non-linear probing
https://github.com/python/cpython/blob/2.7/Objects/dictobject.c#L33-L135

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.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal (except with themselves), and their hash value is derived from their id().

https://docs.python.org/2/glossary.html#term-hashable

```
In [3]: hash(1), hash(1.0)

Out[3]: (1, 1)

In [4]: {(1, 1, 2): 3}

Out[4]: {(1, 1, 2): 3}

In [5]: {{1, 1, 2}: 3}


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-5-e1a507c0aa4a> in <module>()
    ----> 1 {{1, 1, 2}: 3}


        TypeError: unhashable type: 'set'


In [11]: class A(object): pass

         a1 = A()
         a2 = A()
         print({a1: 1, a2:2, a1: 3})
         print(hex(hash(a1)), hex(hash(a2)))
         print(hex(id(a1)), hex(id(a2)))

{<__main__.A object at 0x1089ee410>: 2, <__main__.A object at 0x1084759d0>: 3}
('0x1084759d', '0x1089ee41')
('0x1084759d0', '0x1089ee410')
```

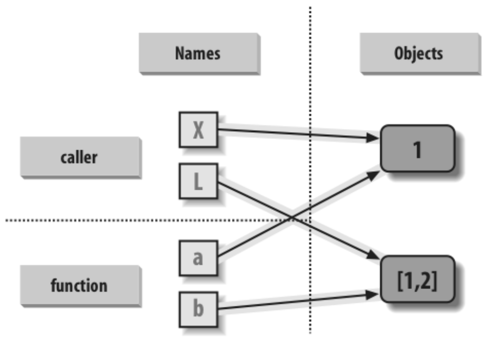
Time Complexity https://wiki.python.org/moin/TimeComplexity

```
In [ ]: Quiz about time complexity.
```

2.3.4 Callable types

These are the types to which the function call operation can be applied: - User-defined functions - User-defined methods in Python 2 & Instance methods in Python 3 - Generator functions - Coroutine functions & Asynchronous generator functions (Python 3 only) - Built-in functions & Built-in methods - Classes (new style classes & classic classes) - Class instances

```
In [12]: def hi():
             return 'Hello world!'

In [21]: class Dog(object):
             def bark(self):
                 return 'woof'

In [16]: def infinite_number_counter():
             i = 0
             while True:
                 yield i
                 i += 1


         c = infinite_number_counter()
         next(c), next(c), next(c), next(c)

Out[16]: (0, 1, 2, 3)

In [19]: l = range(5)
         print(l)
         print(len(l))
         l.append(5)
         print(l)

[0, 1, 2, 3, 4]
5
[0, 1, 2, 3, 4, 5]


In [22]: d = Dog()

In [1]: class A(object):
            def __call__(self, *args, **kwargs):
                print(args, kwargs)

        a = A()
        a(1, foo='bar')

((1,), {'foo': 'bar'})
```



Names and Objects

2.4 Break

2.5 All the variable names in Python are just references

```
X = 1
L = [1, 2]

def foo(a, b):
    pass

foo(X, L)
```

2.6 Naming and binding

For this session, please read the language reference of Python 3, even you are using Python 2. https://docs.python.org/3.6/reference/executionmodel.html#resolution-of-names

2.6.1 Blocks

A block is a piece of Python program text that is executed as a unit. The following are blocks: - a module, a function body, and a class definition. - Each command typed interactively is a block. - A script file (a file given as standard input to the interpreter or specified as a command line argument to the interpreter) is a code block. - A script command (a command specified on the interpreter command line with the '-c' option) is a code block. - The string argument passed to the built-in functions eval() and exec() is a code block.

https://docs.python.org/3.6/reference/executionmodel.html#structure-of-a-program

2.6.2 Binding of names

Names refer to objects. Names are introduced by name binding operations.

The following constructs bind names: - formal parameters to functions, - import statements, - class and function definitions (these bind the class or function name in the defining block), - and targets that are identifiers if occurring in an assignment, - for loop header, - or after as in a with statement or except clause.

The import statement of the form `from ... import *` binds all names defined in the imported module, except those beginning with an underscore. This form may only be used at the module level.

```
In [21]: class A(object):
             import json

         a = A()
         a.json

Out[21]: <module 'json' from '/usr/local/Cellar/python@2/2.7.15/Frameworks/Python.framework/Ve

In [22]: class A(object):
             from json import *

<ipython-input-22-d7ed3184f460>:1: SyntaxWarning: import * only allowed at module level
  class A(object):


In [24]: def foo():
             from json import *

<ipython-input-24-9260272547e5>:1: SyntaxWarning: import * only allowed at module level
  def foo():


In [17]: class A(object):
             for x in [[1],[2]]:
                 pass

         a = A()
         a.x

         b = A()
         b.x.append(10)
         a.x

Out[17]: [2, 10]

In [25]: class A(object):
             try:
                 1/0
             except Exception as e:
                 pass

         a = A()
         a.e

Out[25]: ZeroDivisionError('integer division or modulo by zero')
```

```
In [36]: class A(object):
             with open('x.txt', 'w') as f:
                 pass

         a = A()
         a.f

Out[36]: <closed file 'x.txt', mode 'w' at 0x11067e0c0>
```

A target occurring in a `del` statement is also considered bound for this purpose (though the actual semantics are to unbind the name).

```
In [3]: import dis

        def foo():
            del dis

        print(foo.__code__.co_varnames)
        foo()

('dis',)



        ---------------------------------------------------------------------------

        UnboundLocalError                         Traceback (most recent call last)

        <ipython-input-3-81df8babdcf9> in <module>()
          5
          6 print(foo.__code__.co_varnames)
    ----> 7 foo()


        <ipython-input-3-81df8babdcf9> in foo()
          2
          3 def foo():
    ----> 4     del dis
          5
          6 print(foo.__code__.co_varnames)


        UnboundLocalError: local variable 'dis' referenced before assignment


In [2]: import dis

        def foo():
            global dis
            del dis

        print(foo.__code__.co_varnames)
        foo()

        dis
```

```
()
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-2-643fb00a126a> in <module>()
          8 foo()
          9
    ---> 10 dis


        NameError: name 'dis' is not defined
```

```
In [44]: for i in range(3):
             import json
```

Each assignment or import statement occurs within a block defined by a class or function definition or at the module level (the top-level code block).

If a name is bound in a block, it is a local variable of that block, unless declared as nonlocal or global. If a name is bound at the module level, it is a global variable. (The variables of the module code block are local and global.) If a variable is used in a code block but not defined there, it is a free variable.

Each occurrence of a name in the program text refers to the binding of that name established by the following name resolution rules.

2.6.3 Resolution of names

A scope defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. If the definition occurs in a function block, the scope extends to any blocks contained within the defining one, unless a contained block introduces a different binding for the name.

When a name is used in a code block, it is resolved using the nearest enclosing scope. The set of all such scopes visible to a code block is called the block's environment.

```
In [5]: def foo():
            a = 1
            b = 2
            def bar():
```

```
            a = 10
            print(a, b)
        return bar

    foo()()
```

```
(10, 2)
```

When a name is not found at all, a NameError exception is raised. If the current scope is a function scope, and the name refers to a local variable that has not yet been bound to a value at the point where the name is used, an UnboundLocalError exception is raised. UnboundLocalError is a subclass of NameError.

If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block. This can lead to errors when a name is used within a block before it is bound. This rule is subtle. Python lacks declarations and allows name binding operations to occur anywhere within a code block. The local variables of a code block can be determined by scanning the entire text of the block for name binding operations.

```
In [9]: def foo():
            print(a)

        foo()
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-9-727d9ea5197e> in <module>()
          2     print(a)
          3
    ----> 4 foo()


        <ipython-input-9-727d9ea5197e> in foo()
          1 def foo():
    ----> 2     print(a)
          3
          4 foo()


        NameError: global name 'a' is not defined
```

```
In [7]: def foo():
            del dis

        foo()
```

```
    ---------------------------------------------------------------------------

    UnboundLocalError                         Traceback (most recent call last)

    <ipython-input-7-c61032caeb4a> in <module>()
      2     del dis
      3
----> 4 foo()


    <ipython-input-7-c61032caeb4a> in foo()
      1 def foo():
----> 2     del dis
      3
      4 foo()


    UnboundLocalError: local variable 'dis' referenced before assignment
```

```
In [13]: def foo():
             try:
                 count = 1/0
             except:
                 pass

             print(count)

         foo()
```

```
    ---------------------------------------------------------------------------

    UnboundLocalError                         Traceback (most recent call last)

    <ipython-input-13-ac7601ec0da4> in <module>()
      7     print(count)
      8
----> 9 foo()


    <ipython-input-13-ac7601ec0da4> in foo()
      5         pass
      6
----> 7     print(count)
      8
      9 foo()
```

```
UnboundLocalError: local variable 'count' referenced before assignment
```

If the global statement occurs within a block, all uses of the name specified in the statement refer to the binding of that name in the top-level namespace. Names are resolved in the top-level namespace by searching the global namespace, i.e. the namespace of the module containing the code block, and the builtins namespace, the namespace of the module builtins. The global namespace is searched first. If the name is not found there, the builtins namespace is searched. The global statement must precede all uses of the name.

The global statement has the same scope as a name binding operation in the same block. If the nearest enclosing scope for a free variable contains a global statement, the free variable is treated as a global.

```
In [16]: import dis

         def foo():
             global a

             def bar():
                 print(a)

             return bar
         dis.dis(foo().__code__)
         foo()()
```

```
  7           0 LOAD_GLOBAL              0 (a)
              3 PRINT_ITEM
              4 PRINT_NEWLINE
              5 LOAD_CONST               0 (None)
              8 RETURN_VALUE
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-16-eaa85ba6a4a0> in <module>()
          9     return bar
         10 dis.dis(foo().__code__)
    ---> 11 foo()()


        <ipython-input-16-eaa85ba6a4a0> in bar()
          5
          6     def bar():
    ----> 7         print(a)
```

```
      8
      9     return bar


NameError: global name 'a' is not defined
```

The nonlocal statement causes corresponding names to refer to previously bound variables in the nearest enclosing function scope. SyntaxError is raised at compile time if the given name does not exist in any enclosing function scope.

The namespace for a module is automatically created the first time a module is imported. The main module for a script is always called `__main__`.

```
In [22]: import __main__
         __main__.__name__

Out[22]: '__main__'
```

Class definition blocks and arguments to exec() and eval() are special in the context of name resolution. A class definition is an executable statement that may use and define names. These references follow the normal rules for name resolution with an exception that unbound local variables are looked up in the global namespace. The namespace of the class definition becomes the attribute dictionary of the class. The scope of names defined in a class block is limited to the class block; it does not extend to the code blocks of methods – this includes comprehensions and generator expressions since they are implemented using a function scope. This means that the following will fail:

```
class A:
    a = 42
    b = list(a + i for i in range(10))
```

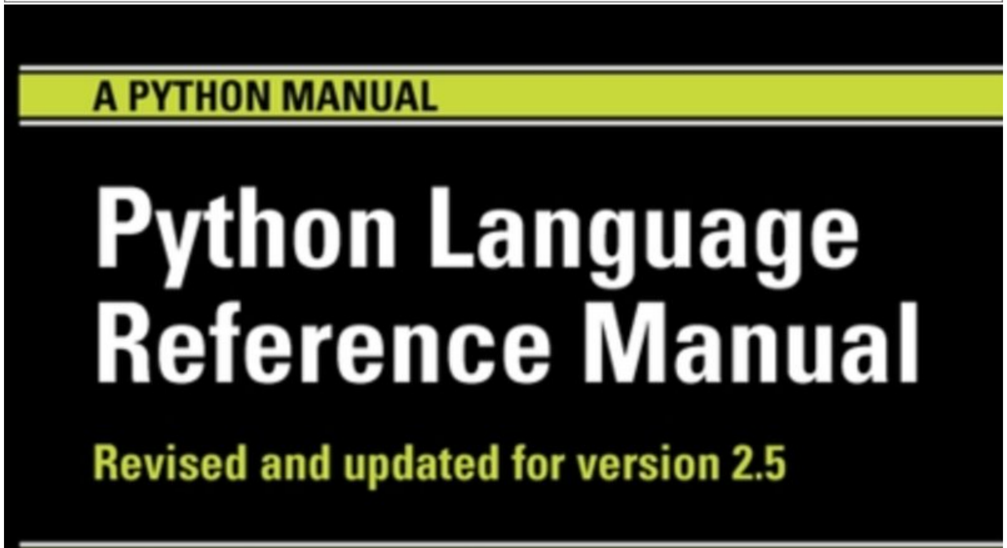
3 References

O'REILLY

Fluent Python

CLEAR, CONCISE, AND EFFECTIVE PROGRAMMING

Luciano Ramalho


A PYTHON MANUAL

Python Language Reference Manual

Revised and updated for version 2.5

4 Videos

```
In [1]: from IPython.display import YouTubeVideo
        # This is a very informative speech, and it worth your watching repeatly.
        YouTubeVideo('7Zlp9rKHGD4')
```

Out[1]:

5 Links

The History of Python - A series of articles on the history of the Python programming language and its community. by Guido van Rossum
A byte of Python

```
In [27]: from decimal import Decimal
         True == 1 == 1+0j == 1.0 == Decimal("1")

Out[27]: True

In [6]: from decimal import Decimal
        {True: 1, 1: 2, 1 + 0j: 3, 1.0: 4, Decimal(1): 5}
```

```
Out[6]: {True: 5}
```

- global https://git.garena.com/beepos/pos_python_server/merge_requests/906/diffs#f5cacdfe36190a6548
- Why should we NOT use sys.setdefaultencoding("utf-8") in a py script? https://stackoverflow.com/questions/3828723/why-should-we-not-use-sys-setdefaultencodingutf-8-in-a-py-script
- http://python-history.blogspot.com/2009/02/early-language-design-and-development.html, pay attention on the integer design and the principles.

Python compare. is, ==, bool