python_introduction

March 12, 2018

1 Welcome to LC102!

- Course overview
- Introduction to Python

1.1 Course overview

1.1.1 Prerequisite

- C programming basics
- Python syntax basics
- Fundamental programming language concepts
  - syntax and semantics
  - typing system
  - programming paradigms

1.1.2 Course Resources

- https://git.garena.com/fanggj/LC102-Python
- Comments
- References

1.1.3 Practice

- Tweak and run the codes by yourself.

If you have any questions in the meantime, feel free to email me at fanggj@seagroup.com with questions.

2 Introduction to Python

2.1 Python Test

2.1.1 Question 1 - What's the output?

```
In [1]: x = 1
```

```
        def foo():
            print(x)

        x = 10

        foo()
```

```
10
```

2.1.2 Question 2 - What's the output?

```
In [2]: def a():
            return []

        def b(x=a()):
            x.append(5)
            print(x)

        b()
        b()
```

```
[5]
[5, 5]
```

2.1.3 Question 3 - What's the output?

```
# q3/a.py
print(__name__)
import a as spam
import b

# q3/b.py
print(__name__)
import a
```

```
In [73]: # run as shell command `python q3/a.py`
         import subprocess
         print(subprocess.check_output("python q3/a.py; exit 0", shell=True, stderr=subprocess
```

```
__main__
a
b
```

2.1.4 Question 4 - What's the output?

```
# q4/a.py
print(__name__)
import a as spam
import b

# q4/b.py
print(__name__)
from a import b
```

```
In [74]: # run as shell command `python q4/a.py`
         import subprocess
         print(subprocess.check_output("python q4/a.py; exit 0", shell=True, stderr=subprocess
```

```
__main__
a
b
Traceback (most recent call last):
  File "q4/a.py", line 2, in <module>
    import a as spam
  File "/Users/fanggj/gitrepo/LC102-Python/lecture00/q4/a.py", line 3, in <module>
    import b
  File "/Users/fanggj/gitrepo/LC102-Python/lecture00/q4/b.py", line 2, in <module>
    from a import b
ImportError: cannot import name b
```

2.1.5 Question 5 - What's the output?

```
In [5]: def foo(*args, **kwargs): pass

        class A(object):
            foo = 1

        print(type(foo))

        a = A()
        print(type(a.foo))

        A.foo = foo
        print(type(a.foo))

        a.foo = foo
        print(type(a.foo))
        print(type(A.foo))
```

```
<type 'function'>
<type 'int'>
```

```
<type 'instancemethod'>
<type 'function'>
<type 'instancemethod'>
```

2.1.6 Python Test Review

```
In [ ]: (0, 'Please take this course.')
        (1, 'Please take this course.')
        (2, 'You can be benefited from this course.')
        (3, 'You can be benefited from this course.')
        (4, 'You can be benefited from this course.')
        (5, 'Pretty good!')
```

2.2 Agenda

- Overview
- Execution Model
- Top-level components

2.3 Overview

2.3.1 Language Perspective

- Interpreted language (Interpreter)
- Readability (Syntax & Pythonic style)

  Braces, brackets, and parentheses https://www.cis.upenn.edu/~matuszek/General/JavaSyntax/parentheses
- Strong, dynamic & duck typing
- Multiple paradigms (OO, procedural, functional)
- Memory management (GC, Reference counting and so on)

With normal typing, suitability is assumed to be determined by an object's type only. In duck typing, an object's suitability is determined by the presence of certain methods and properties (with appropriate meaning), rather than the actual type of the object.

2.3.2 Implementations

- CPython
- PyPy
- Jython
- IronPython

2.3.3 Extension

- Cython

  Cython is an optimising static compiler for both the Python programming language and the extended Cython programming language (based on Pyrex).

2.3.4 Versions

- There are Python2 and Python3
- They are incompatible.
- Fundamental changes:
  - some syntax ('print', 'yield from' ...)
  - implementation details (str, bound methods, dictionary view object ...)
  - ...

str and unicode In February 1991, the code(labeled version 0.9.0) of CPython was published. In October 1991, the first volume of the Unicode standard was published.

2.3.5 Philosophy

```
In [6]: import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

2.3.6 Everything is object.

2.3.7 PyObject & PyVarObject

```
// https://github.com/python/cpython/blob/2.7/Include/object.h#L106
typedef struct _object {
    PyObject_HEAD
} PyObject;
```

```
typedef struct {
    PyObject_VAR_HEAD
} PyVarObject;
```

2.3.8 PyObject_HEAD

```
// https://github.com/python/cpython/blob/2.7/Include/object.h#L78
/* PyObject_HEAD defines the initial segment of every PyObject. */
#define PyObject_HEAD                   \
    _PyObject_HEAD_EXTRA                \
    Py_ssize_t ob_refcnt;               \
    struct _typeobject *ob_type;

// https://github.com/python/cpython/blob/2.7/Include/object.h#L64
#ifdef Py_TRACE_REFS
/* ... */
#else
#define _PyObject_HEAD_EXTRA
#define _PyObject_EXTRA_INIT
#endif
```

2.3.9 PyObject_VAR_HEAD

```
// https://github.com/python/cpython/blob/2.7/Include/object.h#L96
/* PyObject_VAR_HEAD defines the initial segment of all variable-size
 * container objects. */
#define PyObject_VAR_HEAD               \
    PyObject_HEAD                       \
    Py_ssize_t ob_size; /* Number of items in variable part */
```

We will discuss object & type in later lecture.

2.3.10 Overview Recap

Python is an interpreted high-level programming language for general-purpose programming.

Python features a dynamic type system and automatic memory management. It supports multiple programming paradigms, including object-oriented, imperative, functional and procedural, and has a large and comprehensive standard library.

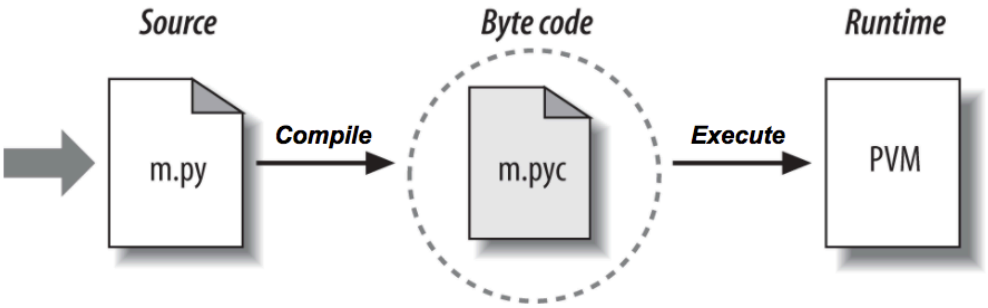
Everything is object in Python.

2.4 Execution Model

2.4.1 How Python runs programs

2.4.2 Essential concepts

- Execution model
  - Code Blocks



how python runs programs

- Execution Frame
- Name
- Scope

2.5 Code blocks

2.5.1 Definition in natural language

A block is a piece of Python program text that is executed as a unit.

The following are blocks: - a module, - a function body, - a class definition, - each command typed interactively, - a script file (standard input or command line argument), - a script command('-c' option), - a string argument passed to the built-in functions eval() and exec(), - an expression read and evaluated by the built-in function input().

2.5.2 Definition in C programming language

```
// https://github.com/python/cpython/blob/2.7/Include/code.h#L9
typedef struct {
    PyObject_HEAD
    int co_argcount;        /* #arguments, except *args */
    int co_nlocals;     /* #local variables */
    int co_stacksize;       /* #entries needed for evaluation stack */
    int co_flags;       /* CO_..., see below */
    PyObject *co_code;      /* instruction opcodes */
    PyObject *co_consts;    /* list (constants used) */
    PyObject *co_names;     /* list of strings (names used) */
    PyObject *co_varnames;  /* tuple of strings (local variable names) */
    PyObject *co_freevars;  /* tuple of strings (free variable names) */
    PyObject *co_cellvars;      /* tuple of strings (cell variable names) */
    /* The rest doesn't count for hash/cmp */
    PyObject *co_filename;  /* string (where it was loaded from) */
    PyObject *co_name;      /* string (name, for reference) */
    int co_firstlineno;     /* first source line number */
    PyObject *co_lnotab;    /* string (encoding addr<->lineno mapping) See Objects/lnotab_note
    void *co_zombieframe;     /* for optimization only (see frameobject.c) */
    PyObject *co_weakreflist;   /* to support weakrefs to code objects */
} PyCodeObject;
```

2.5.3 Useful tools

- dis(module)
  - Brief reference of opcode. https://docs.python.org/2/library/dis.html
- compile(function)
- exec(function or statement)
- inspect(module)

https://stackoverflow.com/questions/12673074/how-should-i-understand-the-output-of-dis-dis

```
# code_block/mymodule.py
class A(object):
    bar = 1


def fib(n):
    if n <= 1:
        return 1
    result = fib(n-1) + fib(n-2)
    return result


def decorator(func):
    x = 10
    def inner(*args, **kwargs):
        print(x)
        return func(*args, **kwargs)
    return inner


x = 1
```

```
In [7]: # code object of a module
        import sys
        from code_block import mymodule
        mymodule??

        import helper
        # Python 3.6: code_block/__pycache__/mymodule.cpython-36.pyc
        # Python 2.x: code_block/mymodule.pyc
        pycfile = 'code_block/mymodule.pyc' if sys.version_info[0] <=2 else 'code_block/__pyca
        module_code = helper.load_code_object_from_pyc(pycfile)

        import dis
        dis.dis(module_code)
```

```
  1           0 LOAD_CONST               0 ('A')
              3 LOAD_NAME                0 (object)
```

```
              6 BUILD_TUPLE              1
              9 LOAD_CONST               1 (<code object A at 0x10bfa0930, file "code_block/myn
             12 MAKE_FUNCTION            0
             15 CALL_FUNCTION            0
             18 BUILD_CLASS
             19 STORE_NAME               1 (A)

  5          22 LOAD_CONST               2 (<code object fib at 0x10bfa05b0, file "code_block/n
             25 MAKE_FUNCTION            0
             28 STORE_NAME               2 (fib)

 12          31 LOAD_CONST               3 (<code object decorator at 0x10bfa0b30, file "code_b
             34 MAKE_FUNCTION            0
             37 STORE_NAME               3 (decorator)

 20          40 LOAD_CONST               4 (1)
             43 STORE_NAME               4 (x)
             46 LOAD_CONST               5 (None)
             49 RETURN_VALUE
```

In [8]:
```
# code object of a function
dis.dis(mymodule.fib.__code__)
```

```
  6           0 LOAD_FAST                0 (n)
              3 LOAD_CONST               1 (1)
              6 COMPARE_OP               1 (<=)
              9 POP_JUMP_IF_FALSE       16

  7          12 LOAD_CONST               1 (1)
             15 RETURN_VALUE

  8     >>   16 LOAD_GLOBAL              0 (fib)
             19 LOAD_FAST                0 (n)
             22 LOAD_CONST               1 (1)
             25 BINARY_SUBTRACT
             26 CALL_FUNCTION            1
             29 LOAD_GLOBAL              0 (fib)
             32 LOAD_FAST                0 (n)
             35 LOAD_CONST               2 (2)
             38 BINARY_SUBTRACT
             39 CALL_FUNCTION            1
             42 BINARY_ADD
             43 STORE_FAST               1 (result)

  9          46 LOAD_FAST                1 (result)
             49 RETURN_VALUE
```

```
In [9]: # code object of a function
        dis.dis(mymodule.decorator.__code__)
```

```
 13           0 LOAD_CONST               1 (10)
              3 STORE_DEREF              1 (x)

 14           6 LOAD_CLOSURE             0 (func)
              9 LOAD_CLOSURE             1 (x)
             12 BUILD_TUPLE              2
             15 LOAD_CONST               2 (<code object inner at 0x10bfa0830, file "code_bloc
             18 MAKE_CLOSURE             0
             21 STORE_FAST               1 (inner)

 17          24 LOAD_FAST                1 (inner)
             27 RETURN_VALUE
```

```
In [10]: # code object of a function
         inner_code = helper.get_object_by_id( ... )
         dis.dis(inner_code)
```

```
 15           0 LOAD_DEREF               1 (x)
              3 PRINT_ITEM
              4 PRINT_NEWLINE

 16           5 LOAD_DEREF               0 (func)
              8 LOAD_FAST                0 (args)
             11 LOAD_FAST                1 (kwargs)
             14 CALL_FUNCTION_VAR_KW     0
             17 RETURN_VALUE
```

```
In [11]: # code object of a class
         class_code = helper.get_object_by_id( ... )
         dis.dis(class_code)
```

```
  1           0 LOAD_NAME                0 (__name__)
              3 STORE_NAME               1 (__module__)

  2           6 LOAD_CONST               0 (1)
              9 STORE_NAME               2 (bar)
             12 LOAD_LOCALS
             13 RETURN_VALUE
```

2.5.4 Naming and binding

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, - and targets that are identifiers if occuring in an assignment, - for loop header, - or after as in a with statement or except clause.

2.5.5 Examples

```
In [12]: helper.print_code_names(module_code)
         mymodule??

{'co_cellvars': (),
 'co_consts': ('A',
               <code object A at 0x10bfa0930, file "code_block/mymodule.py", line 1>,
               <code object fib at 0x10bfa05b0, file "code_block/mymodule.py", line 5>,
               <code object decorator at 0x10bfa0b30, file "code_block/mymodule.py", line 12>,
               1,
               None),
 'co_freevars': (),
 'co_names': ('object', 'A', 'fib', 'decorator', 'x'),
 'co_varnames': ()}


In [13]: helper.print_code_names(mymodule.fib.__code__)

{'co_cellvars': (),
 'co_consts': (None, 1, 2),
 'co_freevars': (),
 'co_names': ('fib',),
 'co_varnames': ('n', 'result')}


In [14]: helper.print_code_names(class_code)

{'co_cellvars': (),
 'co_consts': (1,),
 'co_freevars': (),
 'co_names': ('__name__', '__module__', 'bar'),
 'co_varnames': ()}


In [15]: helper.print_code_names(mymodule.decorator.__code__)

{'co_cellvars': ('func', 'x'),
 'co_consts': (None,
               10,
               <code object inner at 0x10bfa0830, file "code_block/mymodule.py", line 14>),
 'co_freevars': (),
 'co_names': (),
 'co_varnames': ('func', 'inner')}


In [16]: helper.print_code_names(inner_code)

{'co_cellvars': (),
 'co_consts': (None,),
```

```
 'co_freevars': ('func', 'x'),
 'co_names': (),
 'co_varnames': ('args', 'kwargs')}
```

We will discuss naming and binding in later lecture.

2.6 Execution frame

2.6.1 Definition in natural language

A code block is executed in an execution frame.

A frame contains some administrative information(used for debugging) and determines where and how execution continues after the code block's execution has completed.

2.6.2 Definition in C programming language

```c
// https://github.com/python/cpython/blob/2.7/Include/frameobject.h#L16
typedef struct _frame {
    PyObject_VAR_HEAD
    struct _frame *f_back;  /* previous frame, or NULL */
    PyCodeObject *f_code;   /* code segment */
    PyObject *f_builtins;   /* builtin symbol table (PyDictObject) */
    PyObject *f_globals;    /* global symbol table (PyDictObject) */
    PyObject *f_locals;     /* local symbol table (any mapping) */
    PyObject **f_valuestack;    /* points after the last local */
    PyObject **f_stacktop;
    PyObject *f_trace;      /* Trace function */
    PyObject *f_exc_type, *f_exc_value, *f_exc_traceback;
    PyThreadState *f_tstate;
    int f_lasti;        /* Last instruction if called */
    int f_lineno;       /* Current line number */
    int f_iblock;       /* index in f_blockstack */
    PyTryBlock f_blockstack[CO_MAXBLOCKS]; /* for try and loop blocks */
    PyObject *f_localsplus[1];  /* locals+stack, dynamically sized */
} PyFrameObject;
```

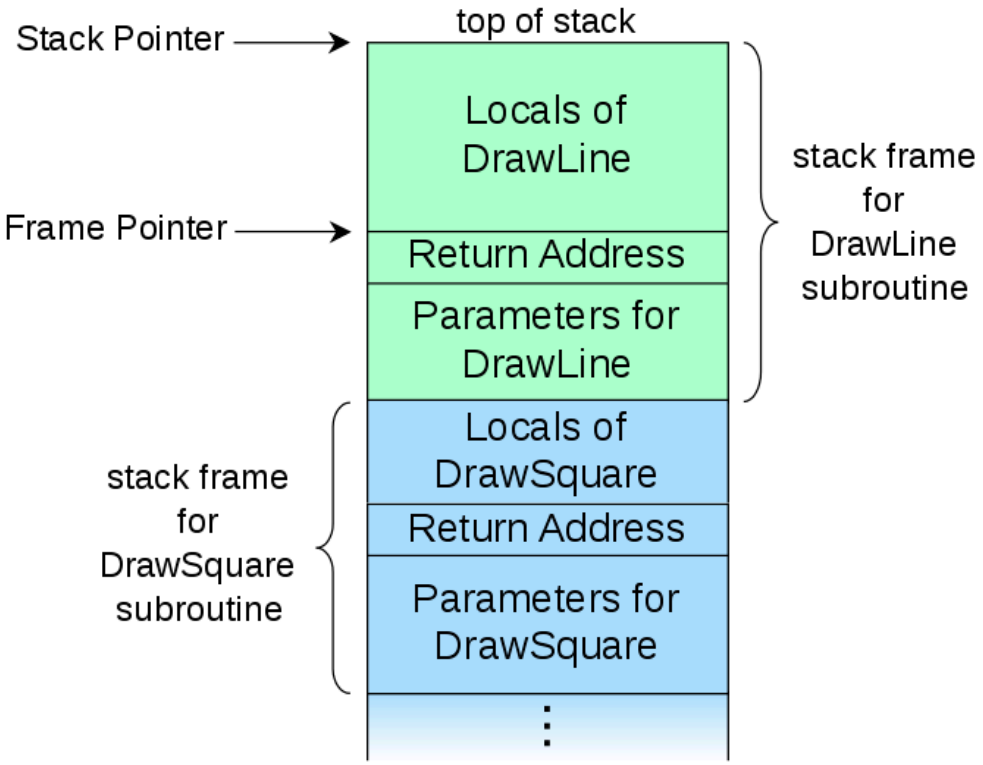
2.6.3 Call stack (C)

2.6.4 Call stack (Python)

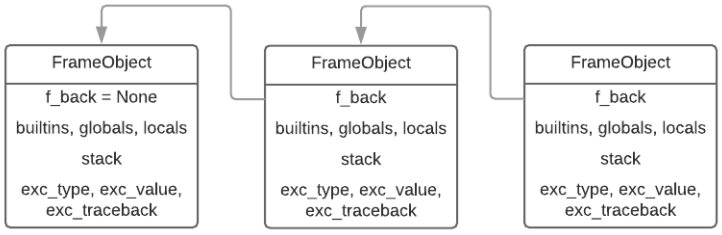
2.6.5 A code block is executed in an execution frame

Proof.

```
In [35]: import inspect
         code = compile("""exc_frame = inspect.currentframe()""", "<string>", "exec")
         exec(code)
         exc_frame.f_code is code
```



typical call stack



call stack python

```
Out[35]: True

In [18]: # frame structure.
         import helper
         helper.print_frame(exc_frame)

[{'back': <frame object at 0x10bfd37f0>},
 {'code': <code object <module> at 0x10bfa0a30, file "<string>", line 1>},
 {'exc_traceback': None, 'exc_type': None, 'exc_value': None},
 {'builtins': 4466004328, 'globals': 4494990984, 'locals': 4494990984},
 {'stack': [('/usr/local/Cellar/python/2.7.14/Frameworks/Python.framework/Versions/2.7/lib/pyth
             174,
             '_run_module_as_main',
             '"__main__", fname, loader, pkg_name)'),
            ('/usr/local/Cellar/python/2.7.14/Frameworks/Python.framework/Versions/2.7/lib/pyth
             72,
             '_run_code',
             'exec code in run_globals'),
            ('/usr/local/lib/python2.7/site-packages/ipykernel_launcher.py',
             16,
             '<module>',
             'app.launch_new_instance()'),
            ('/usr/local/lib/python2.7/site-packages/traitlets/config/application.py',
             658,
             'launch_instance',
             'app.start()'),
            ('/usr/local/lib/python2.7/site-packages/ipykernel/kernelapp.py',
             486,
             'start',
             'self.io_loop.start()'),
            ('/usr/local/lib/python2.7/site-packages/tornado/ioloop.py',
             1065,
             'start',
             'handler_func(fd_obj, events)'),
            ('/usr/local/lib/python2.7/site-packages/tornado/stack_context.py',
             278,
             'null_wrapper',
             '_state.contexts = current_state'),
            ('/usr/local/lib/python2.7/site-packages/zmq/eventloop/zmqstream.py',
             463,
             '_handle_events',
             'raise'),
            ('/usr/local/lib/python2.7/site-packages/zmq/eventloop/zmqstream.py',
             480,
             '_handle_recv',
             'self._run_callback(callback, msg)'),
            ('/usr/local/lib/python2.7/site-packages/zmq/eventloop/zmqstream.py',
             438,
```

```
 '_run_callback',
 'raise'),
('/usr/local/lib/python2.7/site-packages/tornado/stack_context.py',
 278,
 'null_wrapper',
 '_state.contexts = current_state'),
('/usr/local/lib/python2.7/site-packages/ipykernel/kernelbase.py',
 283,
 'dispatcher',
 'return self.dispatch_shell(stream, msg)'),
('/usr/local/lib/python2.7/site-packages/ipykernel/kernelbase.py',
 241,
 'dispatch_shell',
 "self._publish_status(u'idle')"),
('/usr/local/lib/python2.7/site-packages/ipykernel/kernelbase.py',
 421,
 'execute_request',
 'self._abort_queues()'),
('/usr/local/lib/python2.7/site-packages/ipykernel/ipkernel.py',
 258,
 'do_execute',
 'return reply_content'),
('/usr/local/lib/python2.7/site-packages/ipykernel/zmqshell.py',
 537,
 'run_cell',
 'return super(ZMQInteractiveShell, self).run_cell(*args, **kwargs)'),
('/usr/local/lib/python2.7/site-packages/IPython/core/interactiveshell.py',
 2737,
 'run_cell',
 'return result'),
('/usr/local/lib/python2.7/site-packages/IPython/core/interactiveshell.py',
 2850,
 'run_ast_nodes',
 'return False'),
('/usr/local/lib/python2.7/site-packages/IPython/core/interactiveshell.py',
 2902,
 'run_code',
 'return outflag'),
('<ipython-input-17-bef013c441f0>',
 3,
 '<module>',
 u'exec(code)'),
('<string>', 1, '<module>', None)]}]
```

2.6.6 Linked list of frames

```
In [36]: exec("exc_frame = inspect.currentframe()")
         dis.dis(exc_frame.f_code)
```

```
  1           0 LOAD_NAME                0 (inspect)
              3 LOAD_ATTR                1 (currentframe)
              6 CALL_FUNCTION            0
              9 STORE_NAME               2 (exc_frame)
             12 LOAD_CONST               0 (None)
             15 RETURN_VALUE
```

```
In [37]: dis.dis(exc_frame.f_back.f_code)
```

```
  1           0 LOAD_CONST               0 ('exc_frame = inspect.currentframe()')
              3 LOAD_CONST               1 (None)
              6 DUP_TOP
              7 EXEC_STMT
              8 LOAD_CONST               1 (None)
             11 RETURN_VALUE
```

2.7 Control flow & Exceptions

2.7.1 *Exception* Definition in natural language

Exceptions are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions.

I would like to talk about 'the flow of control' which is implemented in same mechanism instead of the exceptions only.

For the simple control flow, please try to compile and disassemble the code by yourself.

2.7.2 PyTryBlock

```
// https://github.com/python/cpython/blob/2.7/Include/frameobject.h#L10
typedef struct {
    int b_type;                 /* what kind of block this is */
    int b_handler;              /* where to jump to find handler */
    int b_level;                /* value stack level to pop to */
} PyTryBlock;
```

2.7.3 PyFrameObject Recap

```
// https://github.com/python/cpython/blob/2.7/Include/frameobject.h#L16
typedef struct _frame {
    /* ... */
    int f_iblock;       /* index in f_blockstack */
    PyTryBlock f_blockstack[CO_MAXBLOCKS]; /* for try and loop blocks */
    /* ... */
} PyFrameObject;
```

2.7.4 PyFrame_BlockSetup

```
// https://github.com/python/cpython/blob/2.7/Objects/frameobject.c#L748
void
PyFrame_BlockSetup(PyFrameObject *f, int type, int handler, int level)
{
    PyTryBlock *b;
    if (f->f_iblock >= CO_MAXBLOCKS)
        Py_FatalError("XXX block stack overflow");
    b = &f->f_blockstack[f->f_iblock++];
    b->b_type = type;
    b->b_level = level;
    b->b_handler = handler;
}
```

2.7.5 for loop

```
In [38]: # SETUP_LOOP: https://github.com/python/cpython/blob/2.7/Python/ceval.c#L2865
         # FOR_ITER: https://github.com/python/cpython/blob/2.7/Python/ceval.c#L2823
         # BREAK_LOOP: https://github.com/python/cpython/blob/2.7/Python/ceval.c#L3248
         import dis
         code = compile("""
         for i in range(10):
             if i < 0:
                 break
             print(i)
         else:
             print("no break")""", "<string>", "exec")
         dis.dis(code)
```

```
  2           0 SETUP_LOOP              46 (to 49)
              3 LOAD_NAME                0 (range)
              6 LOAD_CONST               0 (10)
              9 CALL_FUNCTION            1
             12 GET_ITER
        >>   13 FOR_ITER                27 (to 43)
             16 STORE_NAME               1 (i)

  3          19 LOAD_NAME                1 (i)
             22 LOAD_CONST               1 (0)
             25 COMPARE_OP               0 (<)
             28 POP_JUMP_IF_FALSE       35

  4          31 BREAK_LOOP
             32 JUMP_FORWARD             0 (to 35)

  5     >>   35 LOAD_NAME                1 (i)
             38 PRINT_ITEM
             39 PRINT_NEWLINE
```

```
             40 JUMP_ABSOLUTE           13
        >>   43 POP_BLOCK

  7          44 LOAD_CONST               2 ('no break')
             47 PRINT_ITEM
             48 PRINT_NEWLINE
        >>   49 LOAD_CONST               3 (None)
             52 RETURN_VALUE
```

2.7.6 try ... except statement

```
In [40]: # SETUP_FINALLY, SETUP_EXCEPT: https://github.com/python/cpython/blob/2.7/Python/ceva
         import dis
         code = compile("""try:
             1/0
         except NameError:
             print("should not be NameError")
         except Exception as e:
             print(e)
         finally:
             print("finally end")""", "<string>", "exec")
         dis.dis(code)
```

```
  1          0 SETUP_FINALLY           64 (to 67)
             3 SETUP_EXCEPT            12 (to 18)

  2          6 LOAD_CONST               0 (1)
             9 LOAD_CONST               1 (0)
            12 BINARY_DIVIDE
            13 POP_TOP
            14 POP_BLOCK
            15 JUMP_FORWARD            45 (to 63)

  3     >>   18 DUP_TOP
            19 LOAD_NAME                0 (NameError)
            22 COMPARE_OP              10 (exception match)
            25 POP_JUMP_IF_FALSE       39
            28 POP_TOP
            29 POP_TOP
            30 POP_TOP

  4          31 LOAD_CONST               2 ('should not be NameError')
            34 PRINT_ITEM
            35 PRINT_NEWLINE
            36 JUMP_FORWARD            24 (to 63)

  5     >>   39 DUP_TOP
```

```
             40 LOAD_NAME                1 (Exception)
             43 COMPARE_OP              10 (exception match)
             46 POP_JUMP_IF_FALSE       62
             49 POP_TOP
             50 STORE_NAME               2 (e)
             53 POP_TOP

  6          54 LOAD_NAME                2 (e)
             57 PRINT_ITEM
             58 PRINT_NEWLINE
             59 JUMP_FORWARD             1 (to 63)
       >>    62 END_FINALLY
       >>    63 POP_BLOCK
             64 LOAD_CONST               3 (None)

  8     >>    67 LOAD_CONST               4 ('finally end')
             70 PRINT_ITEM
             71 PRINT_NEWLINE
             72 END_FINALLY
             73 LOAD_CONST               3 (None)
             76 RETURN_VALUE
```

2.7.7 with statement

```
In [39]: # SETUP_WITH: https://github.com/python/cpython/blob/2.7/Python/ceval.c#L2882
         # WITH_CLEANUP: https://github.com/python/cpython/blob/2.7/Python/ceval.c#L2913
         import dis
         code = compile("""with open('q3/a.py') as f:
             print(f.read())""", "<string>", "exec")
         dis.dis(code)

  1           0 LOAD_NAME                0 (open)
              3 LOAD_CONST               0 ('q3/a.py')
              6 CALL_FUNCTION            1
              9 SETUP_WITH              18 (to 30)
             12 STORE_NAME               1 (f)

  2          15 LOAD_NAME                1 (f)
             18 LOAD_ATTR                2 (read)
             21 CALL_FUNCTION            0
             24 PRINT_ITEM
             25 PRINT_NEWLINE
             26 POP_BLOCK
             27 LOAD_CONST               1 (None)
       >>    30 WITH_CLEANUP
             31 END_FINALLY
             32 LOAD_CONST               1 (None)
```

```
 35 RETURN_VALUE
```

2.7.8 Python Exception mechanism

If an exception is raised, Python Interpreter will push the exc_traceback, exc_value and exc_type on the frame stack and handle it by calling block handler. https://github.com/python/cpython/blob/2.7/Python/ceval.c#L3257

- PyEval_EvalFrameEx:
  - PyEval_EvalFrameEx function: https://github.com/python/cpython/blob/2.7/Python/ceval.c#L68
  - How to handle exceptions: https://github.com/python/cpython/blob/2.7/Python/ceval.c#L3257
- PyErr_Fetch: Fetch exception info from ThreadState https://github.com/python/cpython/blob/2.7/Pytho
- set_exc_info: https://github.com/python/cpython/blob/2.7/Python/ceval.c#L3718

2.8 Global interpreter lock(GIL)

```
https://github.com/python/cpython/blob/2.7/Python/ceval.c#L238
static PyThread_type_lock interpreter_lock = 0; /* This is the GIL */

// https://github.com/python/cpython/blob/2.7/Python/ceval.c#L1117
#ifdef WITH_THREAD
            if (interpreter_lock) {
                /* Give another thread a chance */
                if (PyThreadState_Swap(NULL) != tstate)
                    Py_FatalError("ceval: tstate mix-up");
                PyThread_release_lock(interpreter_lock);
                PyThread_acquire_lock(interpreter_lock, 1);
                if (PyThreadState_Swap(tstate) != NULL)
                    Py_FatalError("ceval: orphan tstate");
                /* Check for thread interrupts */
                /* ... */
            }
#endif
```

2.8.1 PyThreadState_Swap

```
// https://github.com/python/cpython/blob/2.7/Python/pystate.c#L336
PyThreadState *
PyThreadState_Swap(PyThreadState *newts)
{
    PyThreadState *oldts = _PyThreadState_Current;

    _PyThreadState_Current = newts;
    /* It should not be possible for more than one thread state
       to be used for a thread.  Check this the best we can in debug
```

```
       builds.
    */
#if defined(Py_DEBUG) && defined(WITH_THREAD)
    if (newts) {
        /* This can be called from PyEval_RestoreThread(). Similar
           to it, we need to ensure errno doesn't change.
        */
        int err = errno;
        PyThreadState *check = PyGILState_GetThisThreadState();
        if (check && check->interp == newts->interp && check != newts)
            Py_FatalError("Invalid thread state for this thread");
        errno = err;
    }
#endif
    return oldts;
}
```

2.8.2 Will one busy thread block others to execute?

```
In [44]: import threading

         def busy_loop():
             print("start loop.")
             for i in xrange(10000000):
                 pass
             print("end loop.")

         def urgent_task():
             print("urgent_task executed.")

         t1 = threading.Thread(target=busy_loop)
         t2 = threading.Thread(target=urgent_task)

         t1.start(), t2.start()

start loop.
urgent_task executed.


Out[44]: (None, None)

end loop.
```

2.8.3 _Py_Ticker & _Py_CheckInterval

```
// https://github.com/python/cpython/blob/2.7/Python/ceval.c#L661
/* for manipulating the thread switch and periodic "stuff" - used to be
   per thread, now just a pair o' globals */
```

```
int _Py_CheckInterval = 100;
volatile int _Py_Ticker = 0; /* so that we hit a "tick" first thing */

// https://github.com/python/cpython/blob/2.7/Python/ceval.c#L1094
        if (--_Py_Ticker < 0) {
            if (*next_instr == SETUP_FINALLY) {
                /* Make the last opcode before
                   a try: finally: block uninterruptible. */
                goto fast_next_opcode;
            }
            _Py_Ticker = _Py_CheckInterval;
            tstate->tick_counter++;
// https://en.wikipedia.org/wiki/Time_Stamp_Counter
#ifdef WITH_TSC
            ticked = 1;
#endif
            if (pendingcalls_to_do) {
                if (Py_MakePendingCalls() < 0) {
                    why = WHY_EXCEPTION;
                    goto on_error;
                }
                if (pendingcalls_to_do)
                    /* MakePendingCalls() didn't succeed.
                       Force early re-execution of this
                       "periodic" code, possibly after
                       a thread switch */
                    _Py_Ticker = 0;
            }
#ifdef WITH_THREAD
            if (interpreter_lock) {
                /* GIL "stuff" */
            }
#endif
        }
```

2.8.4 Execution Model Recap

A block is a piece of Python program text that is executed as a unit.

A code block is executed in an execution frame.

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

The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the global interpreter lock or GIL, that must be held by the current thread before it can safely access Python objects.

In order to emulate concurrency of execution, the interpreter regularly tries to switch threads (see sys.setcheckinterval()). The lock is also released around potentially blocking I/O operations like reading or writing a file, so that other Python threads can run in the meantime.

2.9 Top-level components

2.9.1 Implementation details

https://wiki.python.org/moin/CPythonInterpreterInitialization

A complete Python program is executed in a minimally initialized environment: all built-in and standard modules are available, but none have been initialized, except for sys (various system services), `__builtin__` (built-in functions, exceptions and None) and `__main__`. The latter is used to provide the local and global namespace for execution of the complete program.

```
In [1]: import subprocess
        print(subprocess.check_output("""python -c 'print(globals())'""", shell=True, stderr=su

{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', '__doc__': None, '
```

The interpreter may also be invoked in interactive mode; in this case, it does not read and execute a complete program but reads and executes one statement (possibly compound) at a time. The initial environment is identical to that of a complete program; each statement is executed in the namespace of `__main__`.

```
In [2]: import __main__
        print(__main__.__dict__ is globals())

True
```

```
In [3]: exec("import __main__\nprint(__main__.__dict__ is globals())")

True
```

```
In [4]: import subprocess
        print(subprocess.check_output("""python -c 'import __main__\nprint(__main__.__dict__ is

True
```

Under Unix, a complete program can be passed to the interpreter in three forms: with the -c string command line option, as a file passed as the first command line argument, or as standard input. If the file or standard input is a tty device, the interpreter enters interactive mode; otherwise, it executes the file as a complete program.

```
In [6]: # run as shell command `python q3/a.py`
        import subprocess
        print(subprocess.check_output("echo 'import sys\nprint(sys.version_info)' | python; ex

sys.version_info(major=2, minor=7, micro=14, releaselevel='final', serial=0)
```

```
In [7]: import subprocess
        print(subprocess.check_output("python /dev/ttys000; exit 0", shell=True, stderr=subpro

>>>
```

2.10 References

https://opensource.com/article/17/4/grok-gil

3 Thanks!