python_introduction

March 9, 2018

1 Python Introduction

1.1 Prerequisite

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

1.2 Python Placement Test

1.2.1 Question 1 - What's the output?

```
In [ ]: x = 1

        def foo():
            print(x)

        x = 10

        foo()
```

1.2.2 Question 2 - What's the output?

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

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

        b()
        b()
```

1.2.3 Question 3 - What's the output?

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

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

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

1.2.4 Question 4 - What's the output?

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

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

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

1.2.5 Question 5 - What's the output?

```
In [ ]: 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))
```

1.2.6 Python Placement Test Review

```
In [ ]: def judge(score):
            final_review = 'Please take this course.'
            grade = {
                2: 'You can be benefited from this course.',
                5: 'Pretty good!'
            }

            for required_score, review in grade.items():
                if score < required_score:
                    break
                final_review = review

            return final_review

        for i in range(0, 6):
            print(i, judge(i))
```

1.3 Agenda

- Overview
- How Python runs programs
- Essentials

1.4 Overview

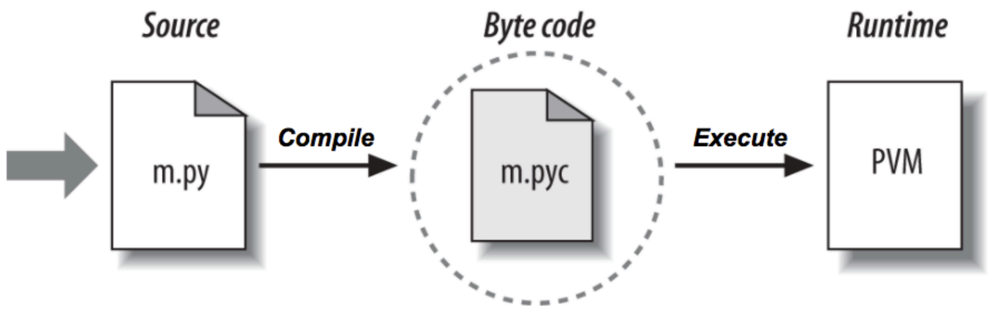
1.4.1 Language Perspective

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

1.4.2 Implementations

- CPython
- PyPy
- Jython
- IronPython



how python runs programs

1.4.3 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.

1.4.4 Philosophy

```
In [ ]: import this
```

1.5 How Python runs programs

1.5.1 Essential concepts

- Execution model
  - Code Blocks
  - Execution Frame
  - Name
  - Scope
- Top-level components

1.6 Code blocks

1.6.1 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().

1.6.2 Definition in C programming language

```
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;
```

1.6.3 Examples

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

```
In [ ]: # code object of a module
        import sys
        from code_block import 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/__pycac
        module_code = helper.load_code_object_from_pyc(pycfile)

        import dis
        dis.dis(module_code)
```

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

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

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

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

1.6.4 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.

1.6.5 Examples

```
In [ ]: helper.print_code_names(module_code)
```

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

```
In [ ]: helper.print_code_names(class_code)
```

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

```
In [ ]: helper.print_code_names(inner_code)
```

We will discuss naming and binding in later lecture.

1.7 Execution frame

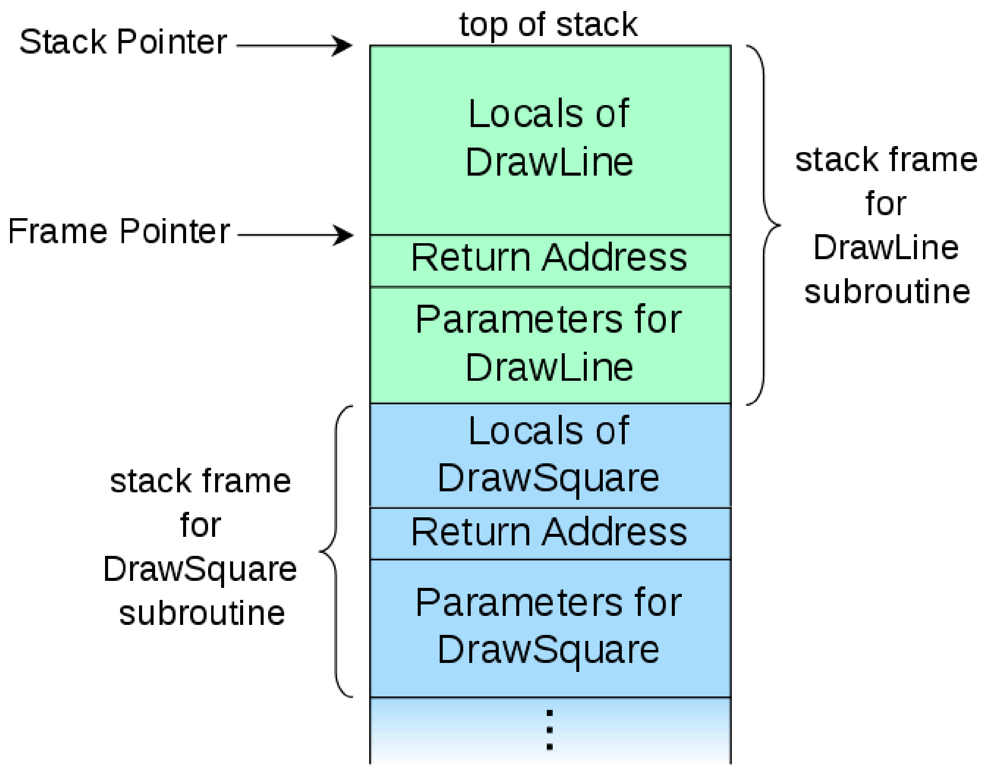
1.7.1 Definition in natural language

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.

1.7.2 Definition in C programming language

```
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 */
```



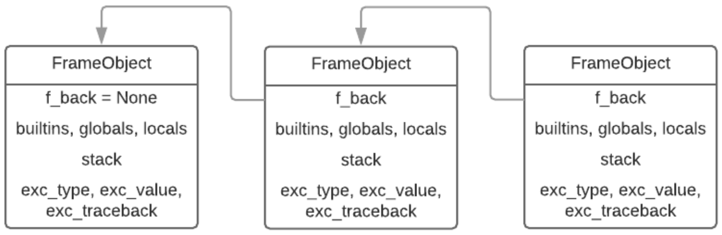
typical call stack

```
    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;
```

1.7.3 Call stack (C)

1.7.4 Call stack (Python)

```
In [ ]: import inspect, dis
        import helper
```


FrameObject
f_back = None
builtins, globals, locals
stack
exc_type, exc_value, exc_traceback
FrameObject
f_back
builtins, globals, locals
stack
exc_type, exc_value, exc_traceback
FrameObject
f_back
builtins, globals, locals
stack
exc_type, exc_value, exc_traceback

call stack python

```
        exec('frame = inspect.currentframe()')
        helper.print_frame(frame)
```

```
In [ ]: dis.dis(frame.f_code)
```

```
In [ ]: dis.dis(frame.f_back.f_code)
```

1.8 Thread

1.9 References

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