왜 컴파일 언어는 빠르고 인터프리터 언어는 느릴까

Python의 내부 구현 분석을 통한 인터프리터 언어의 비용 이해와 C언어와의 구조적 차이

김연하
GDGoC Konkuk 25-26 CORE Member

목차 (Table of Contents)

1. 개요 (Overview)

1-1. 압도적으로 편리한 언어, Python

파이썬을 처음으로 사용했을 때 충격을 받았던 부분

→ **타입을 명시하지 않아도 된다?!!!**

```
def add(a, b):
        return a + b

print(add(40, 2))
print(add("Hello ", "GDGoC!"))
```

← 매우 놀라운 코드임!

1-1. 압도적으로 편리한 언어, Python

파이썬의 편리한 점?

1) 타입을 알리지 않아도 된다.

2) 메모리 관리가 자동으로 이루어진다.

3) 문자열 자료형을 기본 제공한다.

4) 코드 길이가 짧다.

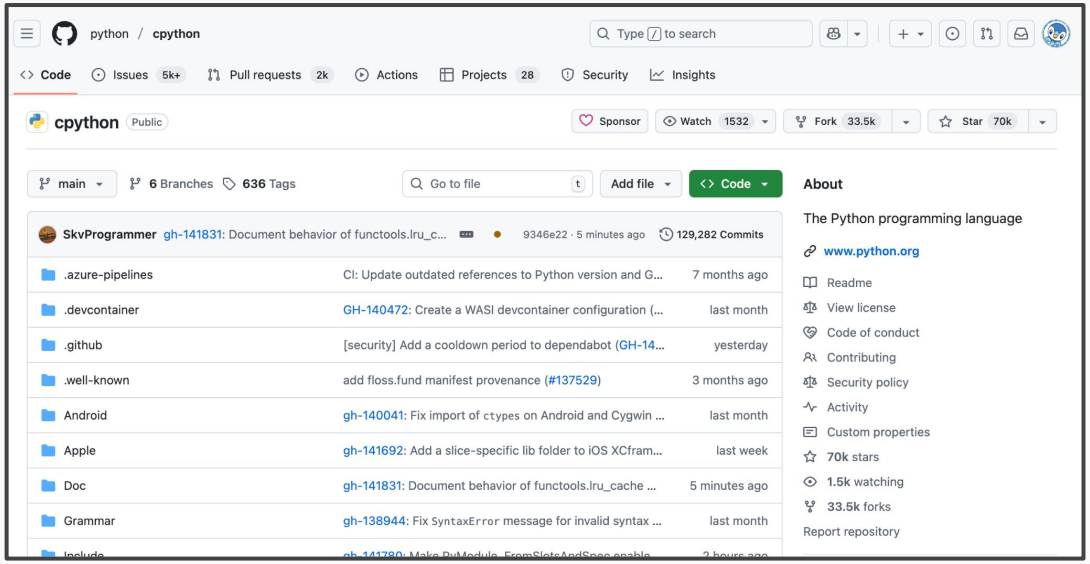
5) 등등 ...

1-2. 이 편리함 뒤에 어떤 비용이 숨어져있을까?

- C언어의 경우, 매번 타입을 알려야 함.
- '프로그래밍 언어'는 결국 인간이 인공적으로 정의 · 설계한 언어이다.
  - 옛날 개발자들이 바보라서 이렇게 언어를 설계한게 아님!
- 근데 Python은 왜 이렇게 편함?

편리함 이면에 어떤 trade-off가 존재할까?

1-2. 이 편리함 뒤에 어떤 비용이 숨어져있을까?



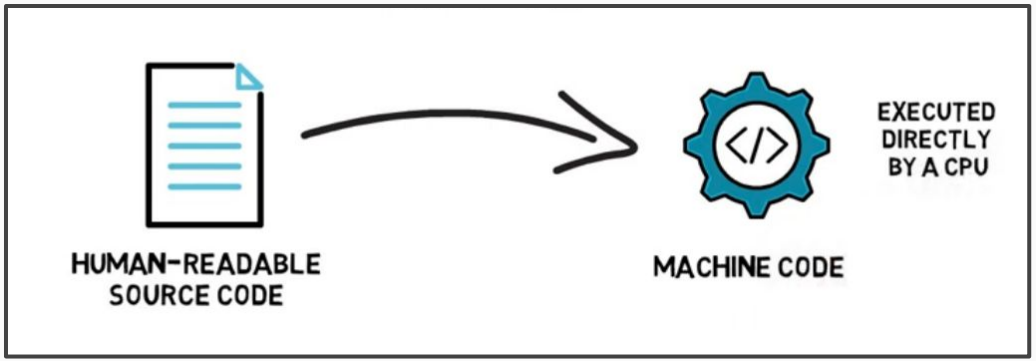
CPython 공식 레포: https://github.com/python/cpython

2. 언어 실행 방식 분류 (Compiled vs Interpreted)

2-1. 컴파일 언어 vs 인터프리터 언어

공통의 목적:

“주어진 소스 코드를 CPU가 실행할 수 있는 형태로 처리한다”

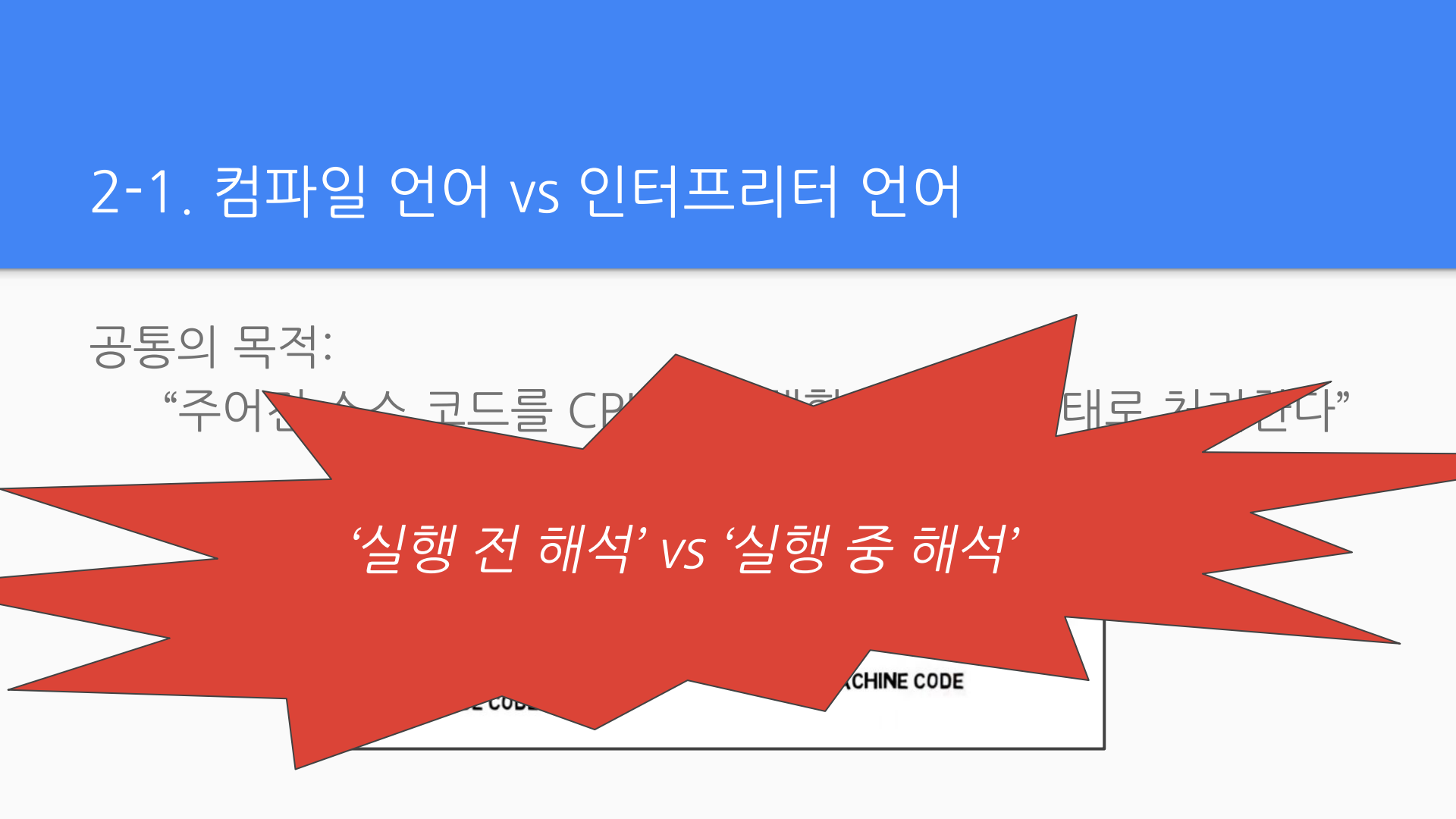


2-1. 컴파일 언어 vs 인터프리터 언어

공통의 목적:

"주어진 소스 코드를 CPU [illegible] 태로 처리한다"

'실행 전 해석' vs '실행 중 해석'



2-1. 컴파일 언어 vs 인터프리터 언어

- **컴파일(Compiled) 언어**
  - “소스 코드를 기계어로 미리 변환해 놓고, 그 결과물을 실행하는 언어”
  - 예시) C/C++, Rust, Go, Swift
- **인터프리터(Interpreted) 언어**
  - “실행 과정에서 소스 코드를 해석하여 즉시 실행하는 언어”
  - 예시) Python, JavaScript, Ruby

2-2. “컴파일 = 빠르다 / 인터프리터 = 느리다” ???

일반적인 인식:

- 컴파일 언어 → 빠름
- 인터프리터 언어 → 느림

2-2. “컴파일 = 빠르다 / 인터프리터 = 느리다” ???

일반적인 인식:

- 컴파일 언어 → 빠름
- 인터프리터 언어 → 느림

“흔한 착각”
속도를 결정하는 실제 요인은
실행 방식 그 자체가 아님!!!

2-2. “컴파일 = 빠르다 / 인터프리터 = 느리다” ???

일반적인 인식:

- 컴파일 언어 → 빠름
- 인터프리터 언어 → 느림

“흔한 착각”
속도를 결정하는 실제 요인은
실행 방식 그 자체가 아님!!!

★언어 성능은 “인터프리터냐 컴파일이냐”가 아니라
“타입 · 메모리 · 디스패치 설계 구조”가 결정★

2-3. 오늘 비교할 대상: “C vs Python” 구조적 차이

- 동적 타입 vs 정적 타입
- 메모리 구조: POD vs Object Model
- 객체 접근 방식: Offset vs Dict
- 동적 디스패치 vs 정적 디스패치
- 최적화 가능 범위

2-3. 오늘 비교할 대상: “C vs Python” 구조적 차이

- 동적 타입 vs 정적 타입
- 메모리 구조: POD vs Object Model
- 객체 접근 방식: Offset vs Dict
- 동적 디스패치 vs 정적 디스패치
- 최적화 가능 범위

※ 현대 언어들은
이 구분을
넘나드는 경우가
흔하다

3. 실행 과정 비교
(Comparison of Execution Models)

3-1. 컴파일 vs 인터프리터 실행 과정 조감도

- **컴파일 언어:** 소스 코드 → 기계어 코드 → 실행 파일 → CPU가 실행
  - 기계어 코드 생성 O
  - CPU가 “기계어 명령어” 단위 실행
- **인터프리터 언어:** 소스 코드 → 바이트 코드 → VM이 실행
  - 기계어 코드 생성 X
  - VM이 “바이트 코드” 단위 실행

3-1. 컴파일 vs 인터프리터 실행 과정 조감도

- **컴파일 언어:** 소스 코드 → 기계어 코드 → 실행 파일 → CPU가 실행
  - 기계어 코드 생성 O
  - CPU가 “기계어 명령어” 단위 실행
- **인터프리터 언어:** 소스 코드 → 바이트 코드 → VM이 실행
  - 기계어 코드 생성 X
  - VM이 “바이트 코드” 단위 실행

3-2. C 코드의 실행 과정

1. **컴파일(Compile)**

   : 소스(.c) → 전처리 → 파싱 → 최적화 → 오브젝트 코드(.o) 생성

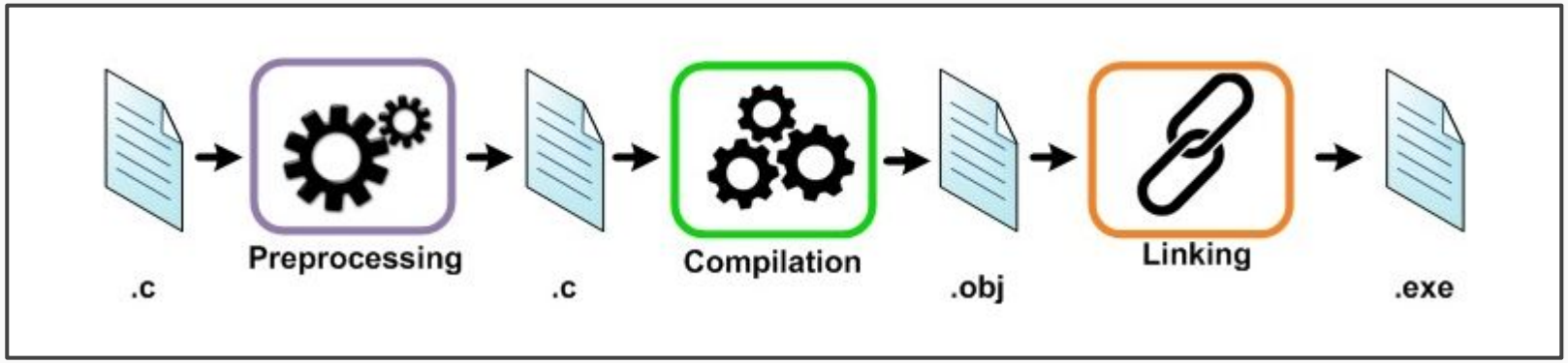
2. **링킹(Link)**

   : 오브젝트 파일 + 라이브러리 결합 → 실행 파일 생성

3. **실행(Run)**

   : CPU가 실행 파일의 기계어를 그대로 수행

3-2. C 코드의 실행 과정



*현대 컴파일러에서, 어셈블리 코드 생성은 옵션

3-3. Python 코드의 실행 과정

1. **파싱(Parser)**

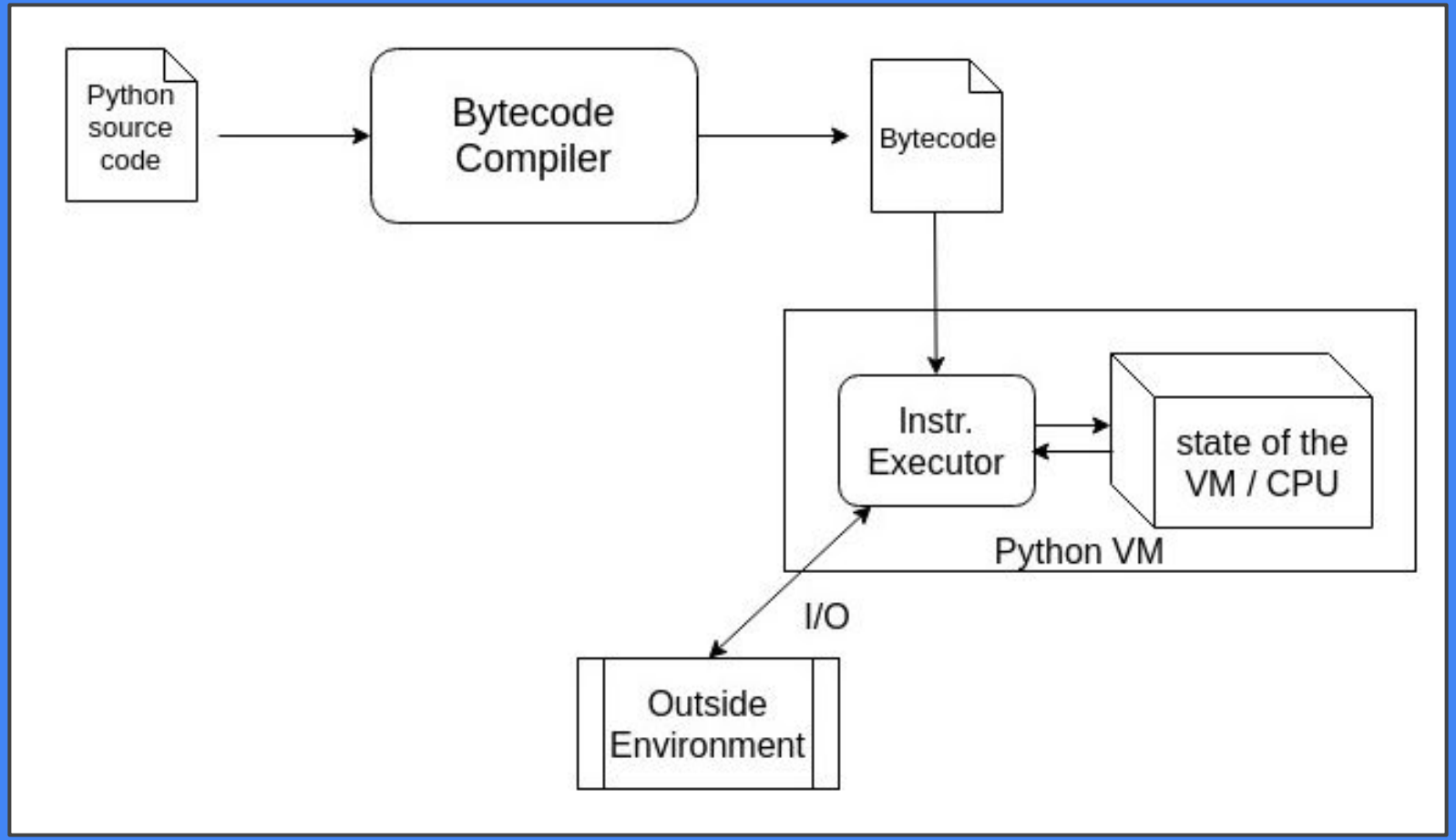
   : 소스를 토큰화 → AST 생성

2. **바이트코드 컴파일(Compile to Bytecode)**

   : AST → VM이 읽을 수 있는 "바이트코드"로 변환

3. **인터프리트(Interpret / Execute)**

   : VM이 ‘바이트코드 해석 실행 Loop’를 돌며 바이트코드를 하나씩 실행


Python source code
Bytecode Compiler
Bytecode
Instr. Executor
state of the VM / CPU
Python VM
I/O
Outside Environment

3-3. Python 코드의 실행 과정

```python
import dis

def add(a, b):
    return a + b

dis.dis(add)
```

dis 라이브러리
: Disassembler for Python bytecode

```
  2           0 LOAD_FAST                0 (a)
              2 LOAD_FAST                1 (b)
              4 BINARY_ADD
              6 RETURN_VALUE
```

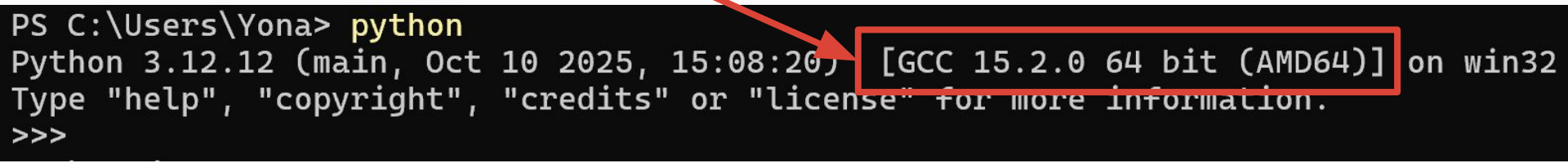
C 코드 → 어셈블리 코드 변환과 유사

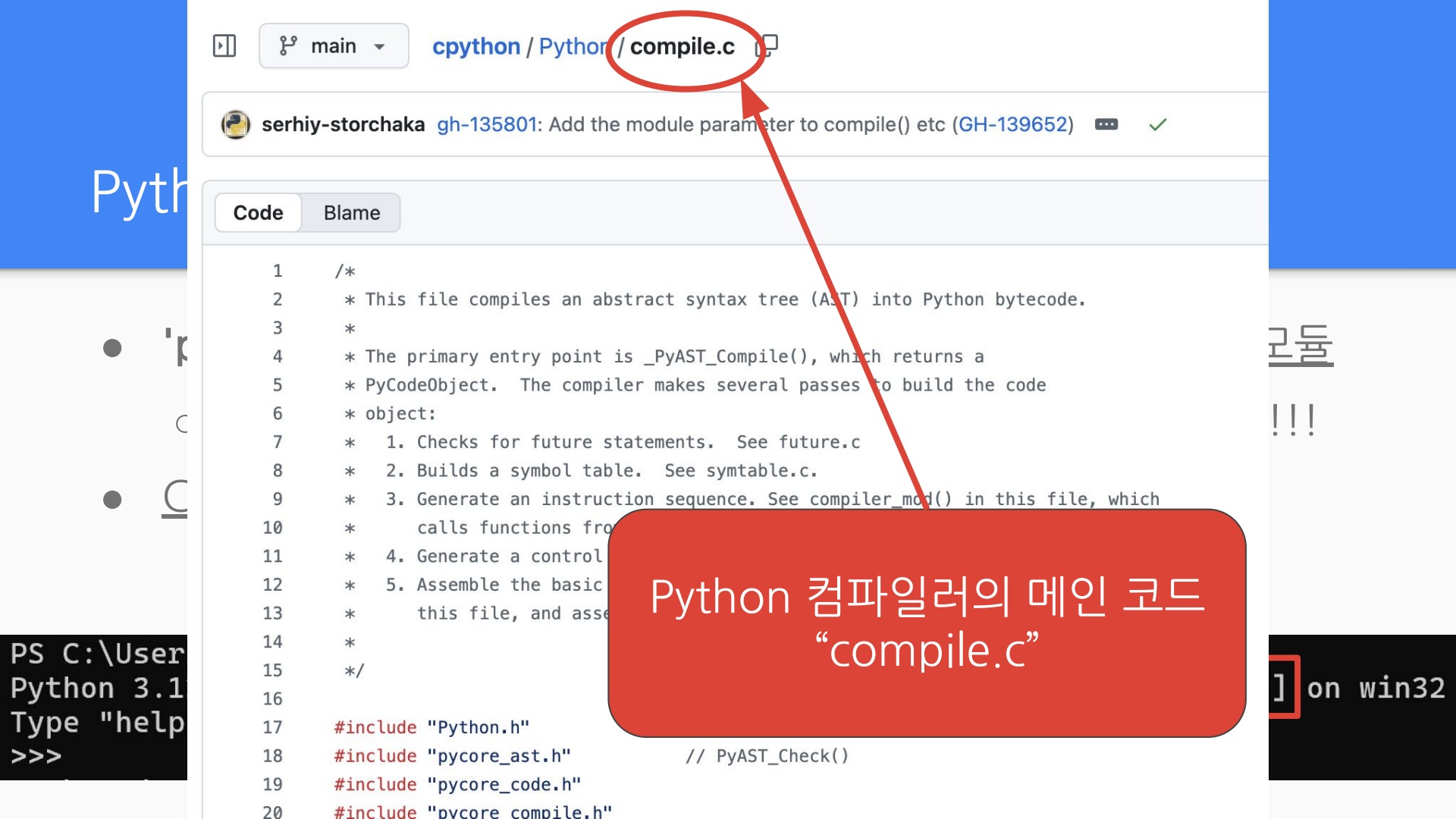
Python Interpreter (= python.exe)

- **'python.exe' 실행 파일** = Python 컴파일러 + VM + 런타임 모듈
  - python 명령어를 입력했을 때 실행되는 바로 그 실행 파일!!!
- C 코드를 C 컴파일러로 빌드해 얻은 결과물

Python Interpreter (= python.exe)

- **'python.exe' 실행 파일** = Python 컴파일러 + VM + 런타임 모듈
  - python 명령어를 입력했을 때 실행되는 바로 그 실행 파일!!!
- C 코드를 C 컴파일러로 빌드해 얻은 결과물




main
cpython / Python / compile.c
serhiy-storchaka gh-135801: Add the module parameter to compile() etc (GH-139652)
Code
Blame
/*
* This file compiles an abstract syntax tree (AST) into Python bytecode.
*
* The primary entry point is _PyAST_Compile(), which returns a
* PyCodeObject. The compiler makes several passes to build the code
* object:
* 1. Checks for future statements. See future.c
* 2. Builds a symbol table. See symtable.c.
* 3. Generate an instruction sequence. See compiler_mod() in this file, which
* 4. Generate a control
* 5. Assemble the basic
*
*/
#include "Python.h"
#include "pycore_ast.h" // PyAST_Check()
#include "pycore_code.h"
#include "pycore_compile.h"
Python 컴파일러의 메인 코드
“compile.c”

main

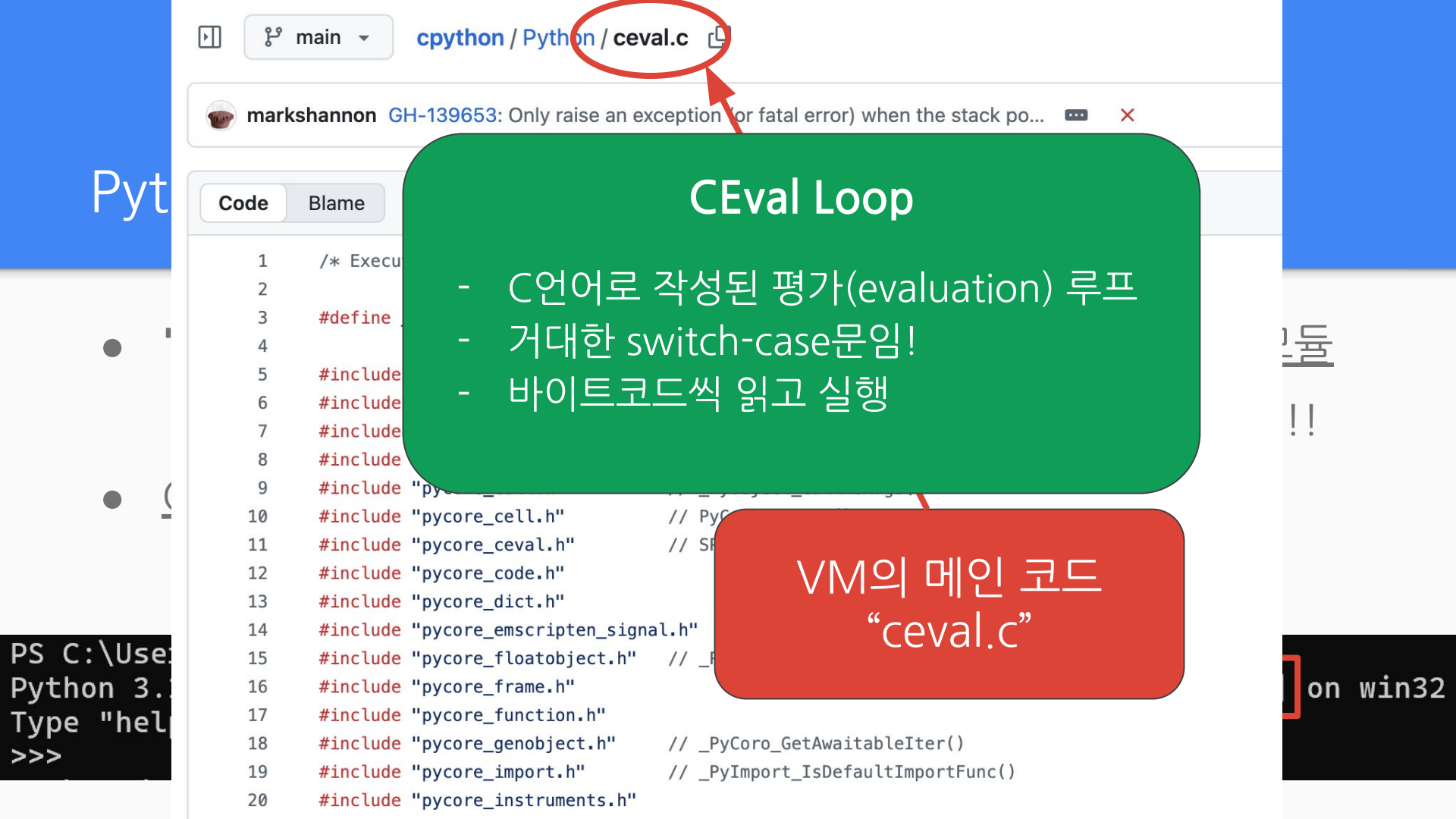
cpython / Python / ceval.c

markshannon GH-139653: Only raise an exception (or fatal error) when the stack po...

Code | Blame

```
1   /* Execute compiled code */
2
3   #define _PY_INTERPRETER
4
5   #include "Python.h"
6   #include "pycore_abstract.h"      // _PyIndex_Check()
7   #include "pycore_audit.h"         // _PySys_Audit()
8   #include "pycore_backoff.h"
9   #include "pycore_call.h"          // _PyObject_CallNoArgs()
10  #include "pycore_cell.h"          // PyC
11  #include "pycore_ceval.h"         // SF
12  #include "pycore_code.h"
13  #include "pycore_dict.h"
14  #include "pycore_emscripten_signal.h"
15  #include "pycore_floatobject.h"   // _F
16  #include "pycore_frame.h"
17  #include "pycore_function.h"
18  #include "pycore_genobject.h"     // _PyCoro_GetAwaitableIter()
19  #include "pycore_import.h"        // _PyImport_IsDefaultImportFunc()
20  #include "pycore_instruments.h"
```

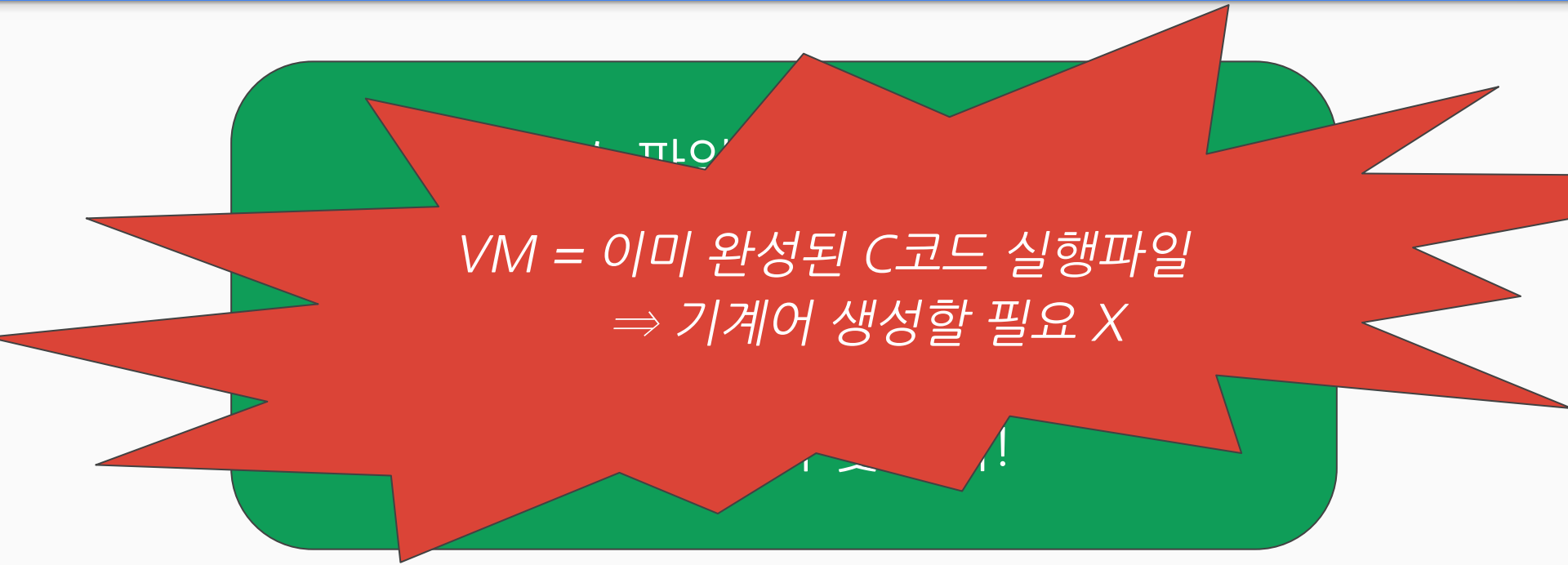
VM의 메인 코드
“ceval.c”


main
cpython / Python / ceval.c
markshannon GH-139653: Only raise an exception (or fatal error) when the stack po...
Code
Blame
CEval Loop
- C언어로 작성된 평가(evaluation) 루프
- 거대한 switch-case문임!
- 바이트코드씩 읽고 실행
VM의 메인 코드
“ceval.c”
1 /* Execu
2
3 #define
4
5 #include
6 #include
7 #include
8 #include
9 #include "py
10 #include "pycore_cell.h" // PyC
11 #include "pycore_ceval.h" // SF
12 #include "pycore_code.h"
13 #include "pycore_dict.h"
14 #include "pycore_emscripten_signal.h"
15 #include "pycore_floatobject.h" // _F
16 #include "pycore_frame.h"
17 #include "pycore_function.h"
18 #include "pycore_genobject.h" // _PyCoro_GetAwaitableIter()
19 #include "pycore_import.h" // _PyImport_IsDefaultImportFunc()
20 #include "pycore_instruments.h"

3-3. Python 코드의 실행 과정

소스 파일을 컴파일하여
Bytecode 더미를 얻고,
VM 프로세스가 루프를 돌며
하나씩 해석 및 처리!

3-3. Python 코드의 실행 과정



3-4. 컴파일 타임과 런타임의 Trade-Off

C언어

- 컴파일 타임에 최대한 많은 것을 결정하고 최적화
- CPU는 별도 추가 연산 없이 주어진 연산을 그대로 실행

⇒ **컴파일 작업↑ 런타임 연산↓**

Python

- 컴파일은 단지 '실행을 위한 준비'

  → 해석 · 최적화 거의 없음
- VM 프로세스가 loop을 돌며 바이트코드를 하나씩 해석 · 실행

⇒ **컴파일 작업↓ 런타임 연산↑**

3-4. 컴파일

컴파일 수행 시간과
실행 시간의 trade-off

C언

- 컴파일 타임에 최대한 많은 것을 결정하고 최적화
- CPU는 별도 추가 연산 없이 주어진 연산을 그대로 실행

⇒ **컴파일 작업↑ 런타임 연산↓**

Python

- 컴파일은 단지 '실행을 위한 준비'
  → 해석 · 최적화 거의 없음
- VM 프로세스가 loop을 돌며 바이트코드를 하나씩 해석 · 실행

⇒ **컴파일 작업↓ 런타임 연산↑**

4. 타입 시스템 & 객체 메모리 구조

(Type system & Object Memory structure)

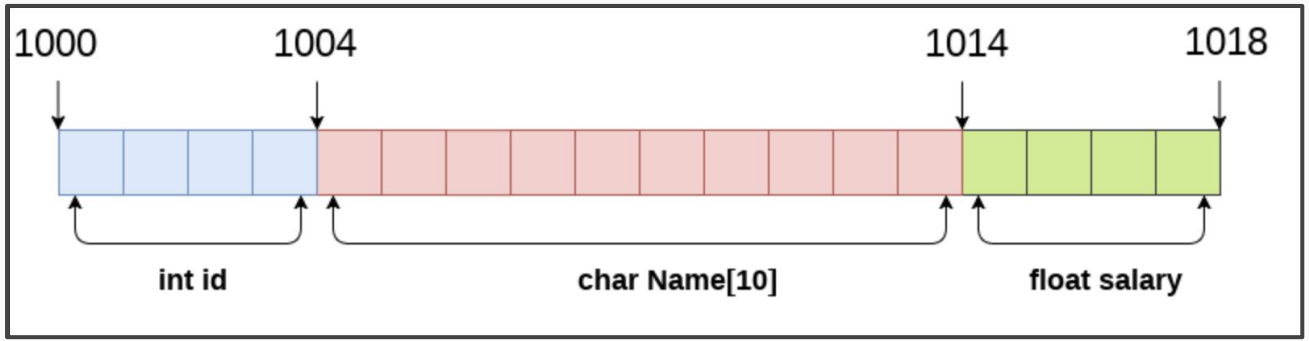
4-1. 정적 타입 vs 동적 타입

- **정적 타입 (Static Type)**
  - 컴파일 타임에 타입 결정
  - 값에 메타데이터 없음 = 순수 데이터(plain data)
- **동적 타입 (Dynamic Type)**
  - 런타임에 타입 결정
  - 값에 메타데이터 포함 = 순수 데이터(plain data) + 타입 관련 정보

4-2. C의 데이터 구조: POD (Plain Old Data)

- 순수한 데이터(plain data)만 존재
- 각 필드(field)가 메모리에 연속적으로 나열됨 → **offset 기반 접근 가능**
- 메모리의 해석 방식이 컴파일 타임에 확정

```
struct Employee
{
    int id;
    char Name[10];
    float salary;
};
```



4-2. C의 데이터 구조: POD (Plain Old Data)

- 순수한 데이터(plain data)만 존재
- 각 필드(field)가 메모리에 연속적으로 나열됨 → **offset 기반 접근 가능**
- 메모리의 해석 방식이 컴파일 타임에 확정

```
struct Employee
{
    int id;
    char Name[10];
    float salary;
};
```

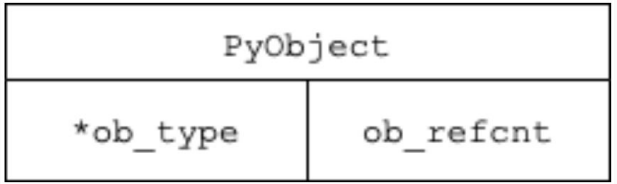
4-3. Python의 철학: “Everything is an object”

- **값, 함수, 타입 모두 객체(object)**
- 모든 객체 → 힙(heap)에 저장
  - 내부 구조의 동적 확장 · 변형 가능
- 모든 변수 → 객체의 참조(reference)
- 모든 연산 → '타입 객체(Type Object)’에 등록된 메서드 호출

4-4. Python Object Model (Cpython)

- PyObject
  - 객체의 '헤더'
  - 모든 객체들은 PyObject를 가짐
- 멤버
  - ob_type: PyTypeObject 객체 ★
  - ob_refcnt: 참조 카운트

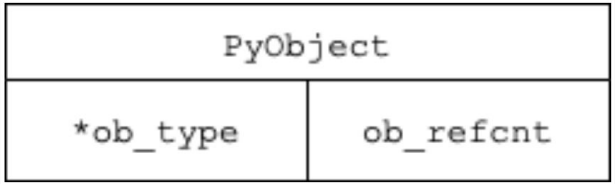
```
// PyObject
struct PyObject {
    Py_ssize_t ob_refcnt;
    PyTypeObject *ob_type;
};
```



4-4. Python Object Model (Cpython)

- PyObject
  - 객체의 '헤더'
  - 모든 객체들은 PyObject를 가짐
- 멤버
  - ob_type: PyTypeObject 객체 ★
  - ob_refcnt: 참조 카운트

```
// PyObject
struct PyObject {
    Py_ssize_t ob_refcnt;
    PyTypeObject *ob_type;
};
```



4-4. Python Object Model (Cpython)

- PyTypeObject
  - ‘타입(type)’ 정보를 저장하는 구조체
- 멤버
  - tp_name: 타입 이름 문자열
  - tp_dict: 파이썬 딕셔너리 객체 ★
  - 그 외 타입 정보들…

```
struct PyTypeObject {
    const char *tp_name; // 타입의 이름
    /* ... */
    PyObject *tp_dict;   // 딕셔너리
};
```

4-4. Python Object Model (Cpython)

- PyTypeObject
  - ‘타입(type)’ 정보를 저장하는 구조체
- 멤버
  - tp_name: 타입 이름 문자열
  - tp_dict: 파이썬 딕셔너리 객체 ★
  - 그 외 타입 정보들…

```
struct PyTypeObject {
    const char *tp_name; // 타입의 이름
    /* ... */
    PyObject *tp_dict;   // 딕셔너리
};
```

4-4. Python Object Model (Cpython)

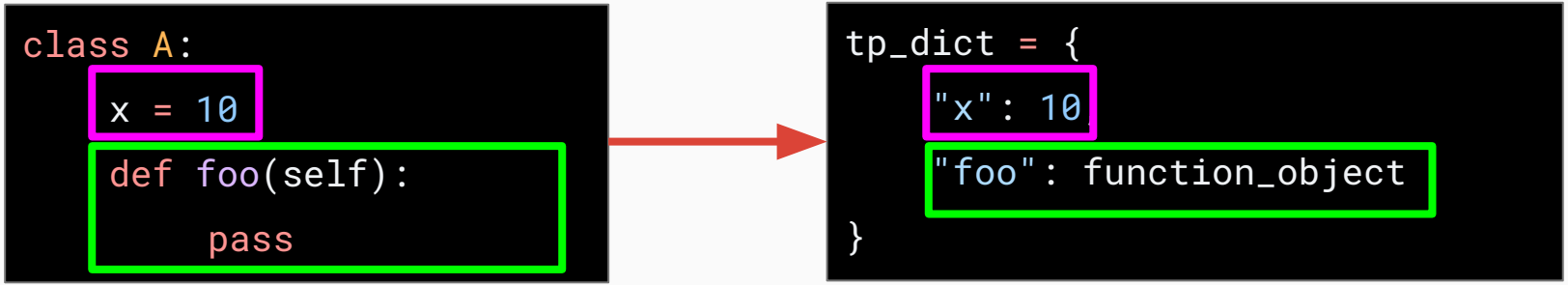
- tp_dict: 파이썬 딕셔너리 객체
  - 객체의 속성(attribute)과 메서드(method)를 저장
  - 해시 테이블(hash table)로 구현됨

```
class A:
    x = 10
    def foo(self):
        pass
```

```
tp_dict = {
    "x": 10,
    "foo": function_object
}
```

4-4. Python Object Model (Cpython)

- tp_dict: 파이썬 딕셔너리 객체
  - 객체의 속성(attribute)과 메서드(method)를 저장
  - 해시 테이블(hash table)로 구현됨



4-4. Python Object Model (Cpython)

1. 딕셔너리 객체 접근 + 해시 테이블 탐색
2. 인스턴스 자신의 dict도 탐색
3. 상속이면 부모 dict까지 연쇄적으로 탐색

```
class A:
    x = 10
    def foo(self):
        pass
```

```
tp_dict = {
    "x": 10,
    "foo": function_object
}
```

4-4. Python Object Model (Cpython)

1. 딕셔너리 객체 접근 + 해시 테이블 탐색
2. 인스턴스 자신의 dict도 탐색
3. 상속이면 부모 dict까지 연쇄적으로 탐색

```
class A:
    x = 10
    def foo(se
        pass
```

속성 · 메서드 접근
비용이 매우 비싸다!!!

```
                                    n object
```

4-5. 동적 타입(Dynamic type)의 구조적 비용

메모리	모든 값, 타입이 PyObject 기반 객체 → 메모리 오버헤드가 크다
시간	모든 연산 · 속성이 동적으로 해석 → 런타임 연산이 많다
최적화	컴파일러가 알 수 있는 정보 제한적 → 컴파일러 최적화가 거의 불가능

5. 디스패치 메커니즘 (Dispatch mechanism)

5-1. Dispatch(디스패치)란

- 실행 대상(코드/함수/속성)을 판정 · 탐색 · 실행하는 전체 과정
  - 판정(Resolution) + 탐색(Lookup) + 호출(Call)
- 판정(Resolution)의 시점에 따른 구분
  - 컴파일 타임 → 정적(static) 디스패치
  - 런타임 → 동적(dynamic) 디스패치

5-1. Dispatch(디스패치)란

- 실행 대상(코드/함수/속성)을 판정 · 탐 · 실행하는 전체 과정
  - 판정(Resoluti 탐 출(Call)
- 판정(Resoluti
  - 컴파일
  - 런타임 dy mic) 디스패치

시간 비용 차이의
주된 원인!!!!

5-2. 정적 디스패치 (Static Dispatch)

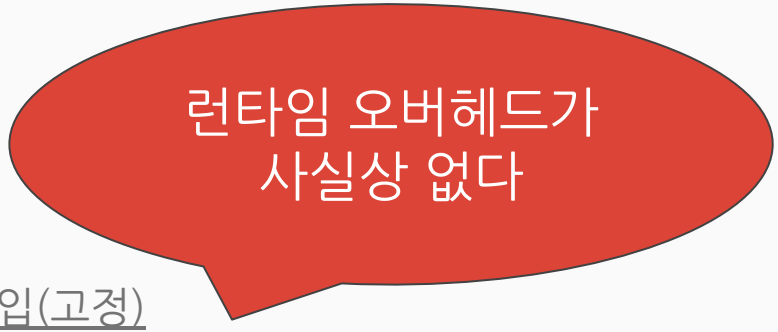
"함수 주소가 이미 기계어에 박혀 있어서 CPU가 바로 jump"

1. **호출 대상 판정 (Compile-time Resolution)**
   : 호출할 함수/메서드를 컴파일 시점에 확정
2. **호출 대상 바인딩(Compile-time Binding)**
   : 선정된 함수의 주소를 기계어 명령어에 그대로 삽입(고정)
3. **기계어 수준에서 직접 호출 (Direct Call)**
   : CPU가 명령어에 고정된 함수 주소로 즉시 jump하여 실행

5-2. 정적 디스패치 (Static Dispatch)

"함수 주소가 이미 기계어에 박혀 있어서 CPU가 바로 jump"

1. **호출 대상 판정 (Compile-time Resolution)**
   : 호출할 함수/메서드를 컴파일 시점에 확정
2. **호출 대상 바인딩(Compile-time Binding)**
   : 선정된 함수의 주소를 기계어 명령어에 그대로 삽입(고정)
3. **기계어 수준에서 직접 호출 (Direct Call)**
   : CPU가 명령어에 고정된 함수 주소로 즉시 jump하여 실행



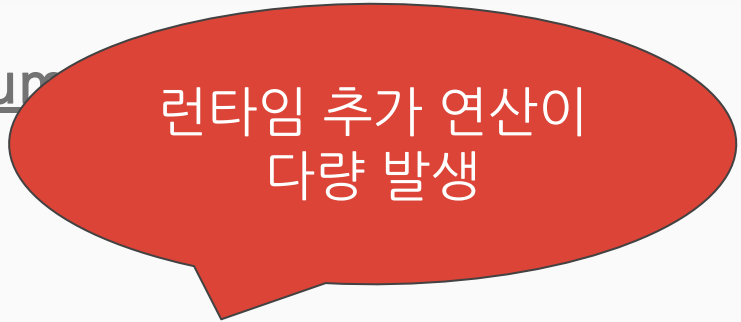
5-3. 동적 디스패치 (Dynamic Dispatch)

"런타임에 호출할 함수의 주소를 찾아온 뒤 jump"

1. **호출 대상 판정 (Run-time Resolution)**
   : 호출할 함수/메서드를 런타임 시점에 확정
2. **호출 대상 탐색 (Runtime Lookup)**
   : 선정된 함수의 주소를 런타임에 획득
3. **간접 호출 (Indirect Call)**
   : 런타임에 획득한 함수 포인터로 indirect jump 수행

5-3. 동적 디스패치 (Dynamic Dispatch)

"런타임에 호출할 함수의 주소를 찾아온 뒤 jum



1. **호출 대상 판정 (Run-time Resolution)**
   : 호출할 함수/메서드를 런타임 시점에 확정
2. **호출 대상 탐색 (Runtime Lookup)**
   : 선정된 함수의 주소를 런타임에 획득
3. **간접 호출 (Indirect Call)**
   : 런타임에 획득한 함수 포인터로 indirect jump 수행

5-3. 동적 디스패치 (Dynamic Dispatch)

"런타임에 호출할 함수의 주소를 찾아온 뒤 jum

1. **호출 대상 판정 (Run-time Resolution)**
   : 호출할 함수/메서드를 런타임 시점에 확정
2. **호출 대상 탐색 (Runtime Lookup)**
   : 선정된 함수의 주소를 런타임에 획득
3. **간접 호출 (Indirect Call)**
   : 런타임에 획득한 함수 포인터로 indirect jump 수행

런타임 추가 연산이
다량 발생

정적 타입 · 동적 타입의
대표 Lookup 전략이 다르다

5-3. 동적 디스패치 (Dynamic Dispatch)

vtable 기반 (C++)

```
1) 객체 → vptr 읽기
2) vptr → vtable 메모리 접근
3) vtable[n] → 함수 주소 로드
4) call [로드한 함수 주소]
```

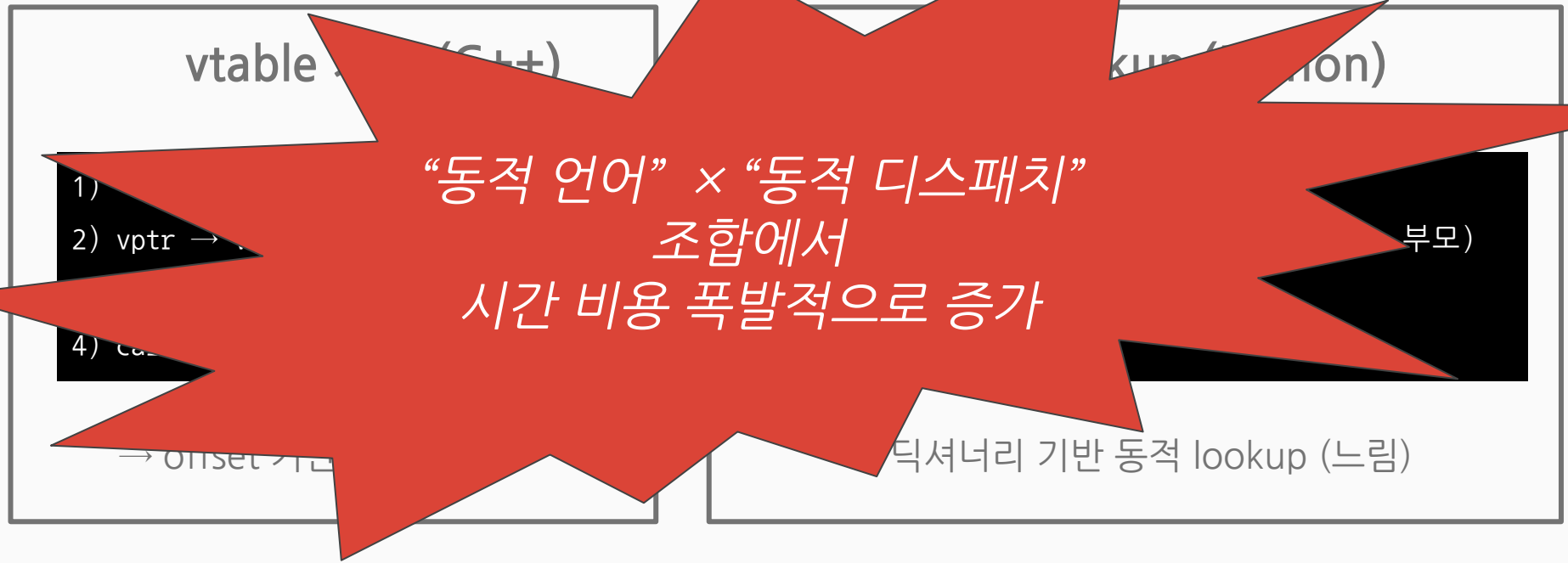
→ offset 기반 접근 (빠름)

Name Lookup (Python)

```
1) 객체 → ob_type 읽기
2) dict 탐색 순서 결정 (인스턴스 → 타입 → 부모)
3) 순서대로 해시 테이블 lookup
4) 찾을 때까지 반복
```

→ 딕셔너리 기반 동적 lookup (느림)

5-3. 동적 디스패치 (Dynamic Dispatch)



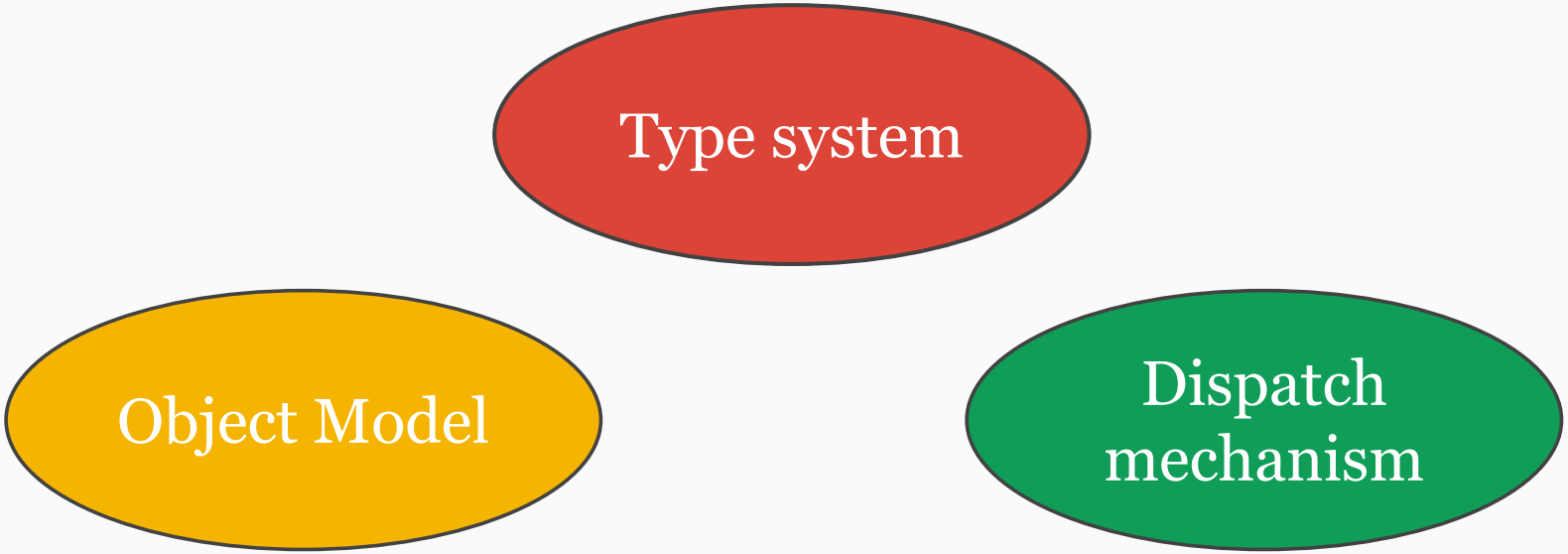
5-4. 정적 디스패치의 최적화(Optimization)

- **인라이닝 (Inlining) ★**
  - 함수 호출 자리에 함수 본문을 그대로 펼쳐 넣음
  - 함수 호출 자체가 없어짐 → 오버헤드 X
  - 여러 추가 최적화가 연쇄적으로 이루어짐

- **분기 예측 최적화 (Branch Prediction)**
  - CPU가 예측하기 쉽도록 분기 구조를 재배치

6. 결론 (Conclusion)

6-1. C vs Python 설계상의 차이



6-2. 컴파일 vs 인터프리터 언어 비교

구분	컴파일 언어 (C/C++)	인터프리터 언어 (Python)
실행 방식	기계어 직접 실행	VM이 바이트코드 해석
실행 준비	컴파일 필요	즉시 실행
성능	고성능	비교적 느림
유연성	정적 타입, 구조 고정	동적 타입, 구조 변경 가능
플랫폼 독립성	플랫폼별 바이너리 필요	어디서나 동일 실행 (VM)

“C is quirky, flawed, and an enormous success.”

“C는 괴팍하고, 결점도 있지만, 거대한 성공이다.”

Dennis Ritchie

“Life is short, you need Python.”

“인생은 짧으니, 당신은 파이썬이 필요하다.”

Bruce Eckel

“No Silver Bullet.”

— 본질적 복잡성(Essence)은 도구로 제거되지 않는다.

감사합니다.

GDGoC Konkuk
25-26 CORE Member
김연하

Email: yhk.dev@gmail.com
Github: github.com/Yonaim
Blog: https://yona.codes

Special Thanks for 25-26 CORE members!