Python behind the scenes #3: stepping through the CPython source code

In the first and the second parts of this series we explored the ideas behind the execution and the compilation of a Python program. We'll continue to focus on ideas in the next parts but this time we'll make an exception and look at the actual code that brings those ideas to life.

Published: Sun 11 October 2020
By Victor Skvortsov
tags: Python behind the scenes Python CPython

Plan for today

The CPython codebase is around 350,000 lines of C code (excluding header files) and almost 600,000 lines of Python code. Undoubtedly, it would be a daunting task to comprehend all of that at once. So, today we'll confine our study to that part of the source code that executes every time we run `python`. We'll start with the `main()` function of the `python` executable and step through the source code until we reach the evaluation loop, a place where the Python bytecode gets executed.

Our goal is not to understand every piece of code we'll encounter but to highlight the most interesting parts, study them, and, eventually, get an approximate idea of what happens at the very start of the execution of a Python program.

There are two more notices I should make. First, we won't step into every function. We'll make only a high-level overview of some parts and dive deep into others. Nevertheless, I promise to present functions in the order of execution. Second, with the exception of a few struct definitions, I'll leave the code as is. The only thing I'll allow myself is to add some comments and to rephrase existing ones. Throughout this post, all multi-line `/**/` comments are original, and all single-line `//` comments are mine. With that said, let's begin our journey through the CPython source code.

Getting CPython

Before we can explore the source code, we need to get it. Let's clone the CPython repository:

```
$ git clone https://github.com/python/cpython/ && cd cpython
```

The current `master` branch is the future CPython 3.10. We're interested in the latest stable release, which is CPython 3.9, so let's switch to the `3.9` branch:

```
$ git checkout 3.9
```

Inside the root directory we find the following contents:

```
$ ls -p
CODE_OF_CONDUCT.md      Objects/                config.sub
Doc/                    PC/                     configure
Grammar/                PCbuild/                configure.ac
Include/                Parser/                 install-sh
LICENSE                 Programs/               m4/
Lib/                    Python/                 netlify.toml
Mac/                    README.rst              pyconfig.h.in
Makefile.pre.in         Tools/                  setup.py
Misc/                   aclocal.m4
Modules/                config.guess
```

Some of the listed subdirectories are of particular importance to us in the course of this series:

- `Grammar/` contains the grammar files we discussed last time.
- `Include/` contains header files. These header files are used both by CPython and by the users of the Python/C API.
- `Lib/` contains standard library modules written in Python. While some modules, such as `argparse` and `wave`, are written in Python entirely, many wrap C code. For example, the Python `io` module wraps the C `_io` module.
- `Modules/` contains standard library modules written in C. While some modules, such as `itertools`, are intended to be imported directly, others are wrapped by the Python modules.
- `Objects/` contains the implementations of the built-in types. If you want to understand how `int` or `list` are implemented, this is the ultimate place to go to.
- `Parser/` contains the old parser, the old parser generator, the new parser and the tokenizer.
- `Programs/` contains source files that are compiled to executables.
- `Python/` contains source files for the interpreter itself. This includes the compiler, the evaluation loop, the `builtins` module and many other interesting things.
- `Tools/` contains tools useful for building and managing CPython. For example, the new parser generator lives here.

If you don't see a directory for tests, and your heart starts beating faster, relax. It's `Lib/test/`. Tests can be useful not only for CPython development but also for getting an understanding of how CPython works. For example, to understand what kinds of optimization the peephole optimizer is expected to do, you can go to `Lib/test/test_peepholer.py` and look. And to understand what some piece of code of the peephole optimizer does, you can delete that piece of code, recompile CPython, run

```
$ ./python.exe -m test test_peepholer
```

and see which tests fail.

In the ideal world, all we need to do to compile CPython is to run `./configure` and `make`:

```
$ ./configure
```

```
$ make -j -s
```

`make` will produce an executable named `python`, but don't be surprised to see `python.exe` on Mac OS. The `.exe` extension is used to distinguish the executable from the `Python/` directory on the case-insensitive filesystem. Check out the Python Developer's Guide for more information on compiling.

At this point we can proudly say that we've built our own copy of CPython:

```
$ ./python.exe
Python 3.9.0+ (heads/3.9-dirty:20bdeedfb4, Oct 10 2020, 16:55:24)
[Clang 10.0.0 (clang-1000.10.44.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 ** 16
65536
```

Source code

The execution of CPython, like execution of any other C program, starts with the `main()` function in `Python/python.c`:

```
/* Minimal main program -- everything is loaded from the library */

#include "Python.h"

#ifdef MS_WINDOWS
int
wmain(int argc, wchar_t **argv)
{
    return Py_Main(argc, argv);
}
#else
int
main(int argc, char **argv)
{
    return Py_BytesMain(argc, argv);
}
#endif
```

There isn't much going on there. The only thing worth mentioning is that on Windows CPython uses `wmain()` instead of `main()` as an entrypoint to receive `argv` as `UTF-16` encoded strings. The affect of this is that on other platforms CPython performs an extra step of converting a `char` string to a `wchar_t` string. The encoding of a `char` string depends on the locale settings, and the encoding of a `wchar_t` string depends on the size of `wchar_t`. For example, if `sizeof(wchar_t) == 4`, the `UCS-4` encoding is used. PEP 383 has more to say on this.

We find `Py_Main()` and `Py_BytesMain()` in `Modules/main.c`. What they do is essentially call `pymain_main()` with slightly different arguments:

```
int
Py_Main(int argc, wchar_t **argv)
{
    _PyArgv args = {
        .argc = argc,
        .use_bytes_argv = 0,
        .bytes_argv = NULL,
        .wchar_argv = argv};
    return pymain_main(&args);
}


int
Py_BytesMain(int argc, char **argv)
{
    _PyArgv args = {
        .argc = argc,
        .use_bytes_argv = 1,
        .bytes_argv = argv,
        .wchar_argv = NULL};
    return pymain_main(&args);
}
```

We should stop on `pymain_main()` for a little bit longer, though, at the first sight, it doesn't seem to do much either:

```
static int
pymain_main(_PyArgv *args)
{
    PyStatus status = pymain_init(args);
    if (_PyStatus_IS_EXIT(status)) {
        pymain_free();
        return status.exitcode;
    }
    if (_PyStatus_EXCEPTION(status)) {
        pymain_exit_error(status);
    }

    return Py_RunMain();
}
```

Last time we learned that before a Python program starts executing, CPython does a lot of things to compile it. It turns out that CPython does a lot of things even before it starts compiling a program. These things constitute the initialization of CPython. As we already said in the first part, CPython works in three stages:

1. initialization
2. compilation; and
3. interpretation.

So, what `pymain_main()` does is call `pymain_init()` to perform the initialization and then call `Py_RunMain()` to proceed with the next stages. The question remains: what does CPython do during the initialization? Let's ponder on this for a moment. At the very least CPython has to:

- find a common language with the OS to handle properly the encodings of arguments, environment variables, standard streams and the file system
- parse the command line arguments and read the environment variables to determine the options to run with
- initialize the runtime state, the main interpreter state and the main thread state
- initialize built-in types and the `builtins` module
- initialize the `sys` module
- set up the import system
- create the `__main__` module.

Before we step into `pymain_init()` to see how all of that is done, let's discuss the initialization process in more detail.

initialization

Starting with CPython 3.8, the initialization is done in three distinct phases:

1. preinitialization
2. core initialization; and
3. main initialization.

The phases gradually introduce new capabilities. The preinitialization phase initializes the runtime state, sets up the default memory allocator and performs very basic configuration. There is no sign of Python yet. The core initialization

phase initializes the main interpreter state and the main thread state, built-in types and exceptions, the `builtins` module, the `sys` module and the import system. At this point, you can use the "core" of Python. However, some things are not available yet. For example, the `sys` module is only partially initialized, and only the import of built-in and frozen modules is supported. After the main initialization phase the CPython is fully initialized and ready to compile and execute a Python program.

What's the benefit of having distinct initialization phases? In a nutshell, it allows to tune CPython more easily. For example, one may set a custom memory allocator in the `preinitialized` state or override the path configuration in the `core_initialized` state. Of course, CPython itself doesn't need to tune anything. Such capabilities are important to the users of the Python/C API who extend and embed Python. PEP 432 and PEP 587 explain in greater detail why having multi-phase initialization is a good idea.

`pymain_init()` mostly deals with the preinitialization and calls `Py_InitializeFromConfig()` in the end to perform the core and the main phases of the initialization:

```
static PyStatus
pymain_init(const _PyArgv *args)
{
    PyStatus status;

    // Initialize the runtime state
    status = _PyRuntime_Initialize();
    if (_PyStatus_EXCEPTION(status)) {
        return status;
    }

    // Initialize default preconfig
    PyPreConfig preconfig;
    PyPreConfig_InitPythonConfig(&preconfig);

    // Perfrom preinitialization
    status = _Py_PreInitializeFromPyArgv(&preconfig, args);
    if (_PyStatus_EXCEPTION(status)) {
        return status;
    }
    // Preinitialized. Prepare config for the next initialization phases

    // Initialize default config
    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    // Store the command line arguments in `config->argv`
    if (args->use_bytes_argv) {
        status = PyConfig_SetBytesArgv(&config, args->argc, args->bytes_argv);
    }
    else {
        status = PyConfig_SetArgv(&config, args->argc, args->wchar_argv);
    }
    if (_PyStatus_EXCEPTION(status)) {
        goto done;
    }

    // Perform core and main initialization
    status = Py_InitializeFromConfig(&config);
        if (_PyStatus_EXCEPTION(status)) {
        goto done;
    }
    status = _PyStatus_OK();

done:
    PyConfig_Clear(&config);
```

```
    return status;
}
```

`_PyRuntime_Initialize()` initializes the runtime state. The runtime state is stored in the global variable called `_PyRuntime` of type `_PyRuntimeState`, which is defined as follows:

```
/* Full Python runtime state */

typedef struct pyruntimestate {
    /* Is running Py_PreInitialize()? */
    int preinitializing;

    /* Is Python preinitialized? Set to 1 by Py_PreInitialize() */
    int preinitialized;

    /* Is Python core initialized? Set to 1 by _Py_InitializeCore() */
    int core_initialized;

    /* Is Python fully initialized? Set to 1 by Py_Initialize() */
    int initialized;

    /* Set by Py_FinalizeEx(). Only reset to NULL if Py_Initialize() is called again. */
    _Py_atomic_address _finalizing;

    struct pyinterpreters {
        PyThread_type_lock mutex;
        PyInterpreterState *head;
        PyInterpreterState *main;
        int64_t next_id;
    } interpreters;

    unsigned long main_thread;

    struct _ceval_runtime_state ceval;
    struct _gilstate_runtime_state gilstate;

    PyPreConfig preconfig;

    // ... less interesting stuff for now
} _PyRuntimeState;
```

The last field `preconfig` of `_PyRuntimeState` holds the configuration that is used to preinitialize CPython. It's also used by the next phase to complete the configuration. Here's the extensively commented definition of `PyPreConfig`:

```
typedef struct {
    int _config_init;     /* _PyConfigInitEnum value */

    /* Parse Py_PreInitializeFromBytesArgs() arguments?
       See PyConfig.parse_argv */
    int parse_argv;

    /* If greater than 0, enable isolated mode: sys.path contains
       neither the script's directory nor the user's site-packages directory.

       Set to 1 by the -I command line option. If set to -1 (default), inherit
       Py_IsolatedFlag value. */
    int isolated;

    /* If greater than 0: use environment variables.
       Set to 0 by -E command line option. If set to -1 (default), it is
       set to !Py_IgnoreEnvironmentFlag. */
    int use_environment;
```

```
        /* Set the LC_CTYPE locale to the user preferred locale? If equals to 0,
           set coerce_c_locale and coerce_c_locale_warn to 0. */
        int configure_locale;

        /* Coerce the LC_CTYPE locale if it's equal to "C"? (PEP 538)

           Set to 0 by PYTHONCOERCECLOCALE=0. Set to 1 by PYTHONCOERCECLOCALE=1.
           Set to 2 if the user preferred LC_CTYPE locale is "C".

           If it is equal to 1, LC_CTYPE locale is read to decide if it should be
           coerced or not (ex: PYTHONCOERCECLOCALE=1). Internally, it is set to 2
           if the LC_CTYPE locale must be coerced.

           Disable by default (set to 0). Set it to -1 to let Python decide if it
           should be enabled or not. */
        int coerce_c_locale;

        /* Emit a warning if the LC_CTYPE locale is coerced?

           Set to 1 by PYTHONCOERCECLOCALE=warn.

           Disable by default (set to 0). Set it to -1 to let Python decide if it
           should be enabled or not. */
        int coerce_c_locale_warn;

    #ifdef MS_WINDOWS
        /* If greater than 1, use the "mbcs" encoding instead of the UTF-8
           encoding for the filesystem encoding.

           Set to 1 if the PYTHONLEGACYWINDOWSFSENCODING environment variable is
           set to a non-empty string. If set to -1 (default), inherit
           Py_LegacyWindowsFSEncodingFlag value.

           See PEP 529 for more details. */
        int legacy_windows_fs_encoding;
    #endif

        /* Enable UTF-8 mode? (PEP 540)

           Disabled by default (equals to 0).

           Set to 1 by "-X utf8" and "-X utf8=1" command line options.
           Set to 1 by PYTHONUTF8=1 environment variable.

           Set to 0 by "-X utf8=0" and PYTHONUTF8=0.

           If equals to -1, it is set to 1 if the LC_CTYPE locale is "C" or
           "POSIX", otherwise it is set to 0. Inherit Py_UTF8Mode value value. */
        int utf8_mode;

        /* If non-zero, enable the Python Development Mode.

           Set to 1 by the -X dev command line option. Set by the PYTHONDEVMODE
           environment variable. */
        int dev_mode;

        /* Memory allocator: PYTHONMALLOC env var.
           See PyMemAllocatorName for valid values. */
        int allocator;
    } PyPreConfig;
```

After the call to `_PyRuntime_Initialize()`, the `_PyRuntime` global variable is initialized to the defaults. Next, `PyPreConfig_InitPythonConfig()` initializes new default `preconfig`, and then `_Py_PreInitializeFromPyArgv()` performs the actual preinitialization. What's the reason to initialize another `preconfig` if there is already one in `_PyRuntime`? Remember

that many functions that CPython calls are also exposed via the Python/C API. So CPython just uses this API in the way it's designed to be used. Another consequence of this is that, when you step through the CPython source code, as we do today, you often encounter functions that seem to do more than you expect them to do. For example, `_PyRuntime_Initialize()` is called several times during the initialization process. Of course it does nothing on the subsequent calls.

`_Py_PreInitializeFromPyArgv()` reads the command line arguments, the environment variables and the global configuration variables to set `_PyRuntime.preconfig`, the current locale and the memory allocator. It doesn't read all the configuration but only those parameters that are relevant to the preinitialization phase. For example, it parses only the `-E -I -X` arguments.

At this point, the runtime is preinitialized. The rest `pymain_init()` does is prepares `config` for the next initialization phase. You should not confuse `config` with `preconfig`. The former is a structure that holds most of the Python configuration. It's heavily used during the initialization stage and then also during the execution of a Python program. To get an idea of what `config` is used for, I recommend you to look over its lengthy definition:

```
/* --- PyConfig ---------------------------------------------- */

typedef struct {
    int _config_init;     /* _PyConfigInitEnum value */

    int isolated;         /* Isolated mode? see PyPreConfig.isolated */
    int use_environment;  /* Use environment variables? see PyPreConfig.use_environment */
    int dev_mode;         /* Python Development Mode? See PyPreConfig.dev_mode */

    /* Install signal handlers? Yes by default. */
    int install_signal_handlers;

    int use_hash_seed;      /* PYTHONHASHSEED=x */
    unsigned long hash_seed;

    /* Enable faulthandler?
       Set to 1 by -X faulthandler and PYTHONFAULTHANDLER. -1 means unset. */
    int faulthandler;

    /* Enable PEG parser?
       1 by default, set to 0 by -X oldparser and PYTHONOLDPARSER */
    int _use_peg_parser;

    /* Enable tracemalloc?
       Set by -X tracemalloc=N and PYTHONTRACEMALLOC. -1 means unset */
    int tracemalloc;

    int import_time;        /* PYTHONPROFILEIMPORTTIME, -X importtime */
    int show_ref_count;     /* -X showrefcount */
    int dump_refs;          /* PYTHONDUMPREFS */
    int malloc_stats;       /* PYTHONMALLOCSTATS */

    /* Python filesystem encoding and error handler:
       sys.getfilesystemencoding() and sys.getfilesystemencodeerrors().

       Default encoding and error handler:

       * if Py_SetStandardStreamEncoding() has been called: they have the
         highest priority;
       * PYTHONIOENCODING environment variable;
       * The UTF-8 Mode uses UTF-8/surrogateescape;
       * If Python forces the usage of the ASCII encoding (ex: C locale
         or POSIX locale on FreeBSD or HP-UX), use ASCII/surrogateescape;
       * locale encoding: ANSI code page on Windows, UTF-8 on Android and
         VxWorks, LC_CTYPE locale encoding on other platforms;
       * On Windows, "surrogateescape" error handler;
```

```
   * "surrogateescape" error handler if the LC_CTYPE locale is "C" or "POSIX";
   * "surrogateescape" error handler if the LC_CTYPE locale has been coerced
     (PEP 538);
   * "strict" error handler.

   Supported error handlers: "strict", "surrogateescape" and
   "surrogatepass". The surrogatepass error handler is only supported
   if Py_DecodeLocale() and Py_EncodeLocale() use directly the UTF-8 codec;
   it's only used on Windows.

   initfsencoding() updates the encoding to the Python codec name.
   For example, "ANSI_X3.4-1968" is replaced with "ascii".

   On Windows, sys._enablelegacywindowsfsencoding() sets the
   encoding/errors to mbcs/replace at runtime.


   See Py_FileSystemDefaultEncoding and Py_FileSystemDefaultEncodeErrors.
   */
wchar_t *filesystem_encoding;
wchar_t *filesystem_errors;

wchar_t *pycache_prefix;  /* PYTHONPYCACHEPREFIX, -X pycache_prefix=PATH */
int parse_argv;           /* Parse argv command line arguments? */

/* Command line arguments (sys.argv).

   Set parse_argv to 1 to parse argv as Python command line arguments
   and then strip Python arguments from argv.

   If argv is empty, an empty string is added to ensure that sys.argv
   always exists and is never empty. */
PyWideStringList argv;

/* Program name:

   - If Py_SetProgramName() was called, use its value.
   - On macOS, use PYTHONEXECUTABLE environment variable if set.
   - If WITH_NEXT_FRAMEWORK macro is defined, use __PYVENV_LAUNCHER__
     environment variable is set.
   - Use argv[0] if available and non-empty.
   - Use "python" on Windows, or "python3 on other platforms. */
wchar_t *program_name;

PyWideStringList xoptions;     /* Command line -X options */

/* Warnings options: lowest to highest priority. warnings.filters
   is built in the reverse order (highest to lowest priority). */
PyWideStringList warnoptions;

/* If equal to zero, disable the import of the module site and the
   site-dependent manipulations of sys.path that it entails. Also disable
   these manipulations if site is explicitly imported later (call
   site.main() if you want them to be triggered).

   Set to 0 by the -S command line option. If set to -1 (default), it is
   set to !Py_NoSiteFlag. */
int site_import;

/* Bytes warnings:

   * If equal to 1, issue a warning when comparing bytes or bytearray with
     str or bytes with int.
   * If equal or greater to 2, issue an error.

   Incremented by the -b command line option. If set to -1 (default), inherit
```

```
   Py_BytesWarningFlag value. */
int bytes_warning;

/* If greater than 0, enable inspect: when a script is passed as first
   argument or the -c option is used, enter interactive mode after
   executing the script or the command, even when sys.stdin does not appear
   to be a terminal.

   Incremented by the -i command line option. Set to 1 if the PYTHONINSPECT
   environment variable is non-empty. If set to -1 (default), inherit
   Py_InspectFlag value. */
int inspect;

/* If greater than 0: enable the interactive mode (REPL).

   Incremented by the -i command line option. If set to -1 (default),
   inherit Py_InteractiveFlag value. */
int interactive;

/* Optimization level.

   Incremented by the -O command line option. Set by the PYTHONOPTIMIZE
   environment variable. If set to -1 (default), inherit Py_OptimizeFlag
   value. */
int optimization_level;

/* If greater than 0, enable the debug mode: turn on parser debugging
   output (for expert only, depending on compilation options).

   Incremented by the -d command line option. Set by the PYTHONDEBUG
   environment variable. If set to -1 (default), inherit Py_DebugFlag
   value. */
int parser_debug;

/* If equal to 0, Python won't try to write ``.pyc`` files on the
   import of source modules.

   Set to 0 by the -B command line option and the PYTHONDONTWRITEBYTECODE
   environment variable. If set to -1 (default), it is set to
   !Py_DontWriteBytecodeFlag. */
int write_bytecode;

/* If greater than 0, enable the verbose mode: print a message each time a
   module is initialized, showing the place (filename or built-in module)
   from which it is loaded.

   If greater or equal to 2, print a message for each file that is checked
   for when searching for a module. Also provides information on module
   cleanup at exit.

   Incremented by the -v option. Set by the PYTHONVERBOSE environment
   variable. If set to -1 (default), inherit Py_VerboseFlag value. */
int verbose;

/* If greater than 0, enable the quiet mode: Don't display the copyright
   and version messages even in interactive mode.

   Incremented by the -q option. If set to -1 (default), inherit
   Py_QuietFlag value. */
int quiet;

/* If greater than 0, don't add the user site-packages directory to
   sys.path.

   Set to 0 by the -s and -I command line options , and the PYTHONNOUSERSITE
   environment variable. If set to -1 (default), it is set to
```

```
          !Py_NoUserSiteDirectory. */
        int user_site_directory;

        /* If non-zero, configure C standard steams (stdio, stdout,
           stderr):

           - Set O_BINARY mode on Windows.
           - If buffered_stdio is equal to zero, make streams unbuffered.
             Otherwise, enable streams buffering if interactive is non-zero. */
        int configure_c_stdio;

        /* If equal to 0, enable unbuffered mode: force the stdout and stderr
           streams to be unbuffered.

           Set to 0 by the -u option. Set by the PYTHONUNBUFFERED environment
           variable.
           If set to -1 (default), it is set to !Py_UnbufferedStdioFlag. */
        int buffered_stdio;

        /* Encoding of sys.stdin, sys.stdout and sys.stderr.
           Value set from PYTHONIOENCODING environment variable and
           Py_SetStandardStreamEncoding() function.
           See also 'stdio_errors' attribute. */
        wchar_t *stdio_encoding;

        /* Error handler of sys.stdin and sys.stdout.
           Value set from PYTHONIOENCODING environment variable and
           Py_SetStandardStreamEncoding() function.
           See also 'stdio_encoding' attribute. */
        wchar_t *stdio_errors;

    #ifdef MS_WINDOWS
        /* If greater than zero, use io.FileIO instead of WindowsConsoleIO for sys
           standard streams.

           Set to 1 if the PYTHONLEGACYWINDOWSSTDIO environment variable is set to
           a non-empty string. If set to -1 (default), inherit
           Py_LegacyWindowsStdioFlag value.

           See PEP 528 for more details. */
        int legacy_windows_stdio;
    #endif

        /* Value of the --check-hash-based-pycs command line option:

           - "default" means the 'check_source' flag in hash-based pycs
             determines invalidation
           - "always" causes the interpreter to hash the source file for
             invalidation regardless of value of 'check_source' bit
           - "never" causes the interpreter to always assume hash-based pycs are
             valid

           The default value is "default".

           See PEP 552 "Deterministic pycs" for more details. */
        wchar_t *check_hash_pycs_mode;

        /* --- Path configuration inputs ------------ */

        /* If greater than 0, suppress _PyPathConfig_Calculate() warnings on Unix.
           The parameter has no effect on Windows.

           If set to -1 (default), inherit !Py_FrozenFlag value. */
        int pathconfig_warnings;

        wchar_t *pythonpath_env; /* PYTHONPATH environment variable */
```

```
    wchar_t *home;          /* PYTHONHOME environment variable,
                               see also Py_SetPythonHome(). */

    /* --- Path configuration outputs ----------- */

    int module_search_paths_set;  /* If non-zero, use module_search_paths */
    PyWideStringList module_search_paths;  /* sys.path paths. Computed if
                                      module_search_paths_set is equal
                                      to zero. */

    wchar_t *executable;        /* sys.executable */
    wchar_t *base_executable;   /* sys._base_executable */
    wchar_t *prefix;            /* sys.prefix */
    wchar_t *base_prefix;       /* sys.base_prefix */
    wchar_t *exec_prefix;       /* sys.exec_prefix */
    wchar_t *base_exec_prefix;  /* sys.base_exec_prefix */
    wchar_t *platlibdir;        /* sys.platlibdir */

    /* --- Parameter only used by Py_Main() ---------- */

    /* Skip the first line of the source ('run_filename' parameter), allowing use of non-Unix forms of
       "#!cmd".  This is intended for a DOS specific hack only.

       Set by the -x command line option. */
    int skip_source_first_line;

    wchar_t *run_command;   /* -c command line argument */
    wchar_t *run_module;    /* -m command line argument */
    wchar_t *run_filename;  /* Trailing command line argument without -c or -m */

    /* --- Private fields ---------------------------- */

    /* Install importlib? If set to 0, importlib is not initialized at all.
       Needed by freeze_importlib. */
    int _install_importlib;

    /* If equal to 0, stop Python initialization before the "main" phase */
    int _init_main;

    /* If non-zero, disallow threads, subprocesses, and fork.
       Default: 0. */
    int _isolated_interpreter;

    /* Original command line arguments. If _orig_argv is empty and _argv is
       not equal to [''], PyConfig_Read() copies the configuration 'argv' list
       into '_orig_argv' list before modifying 'argv' list (if parse_argv
       is non-zero).

       _PyConfig_Write() initializes Py_GetArgcArgv() to this list. */
    PyWideStringList _orig_argv;
} PyConfig;
```

The same way as `pymain_init()` called `PyPreConfig_InitPythonConfig()` to create default `preconfig`, it now calls `PyConfig_InitPythonConfig()` to create default `config`. It then calls `PyConfig_SetBytesArgv()` to store the command line arguments in `config.argv` and `Py_InitializeFromConfig()` to perform the core and the main initialization phases. We move further from `pymain_init()` to `Py_InitializeFromConfig()`:

```
PyStatus
Py_InitializeFromConfig(const PyConfig *config)
{
    if (config == NULL) {
        return _PyStatus_ERR("initialization config is NULL");
    }
```

```
    PyStatus status;

    // Yeah, call once again
    status = _PyRuntime_Initialize();
    if (_PyStatus_EXCEPTION(status)) {
        return status;
    }
    _PyRuntimeState *runtime = &_PyRuntime;

    PyThreadState *tstate = NULL;
    // The core initialization phase
    status = pyinit_core(runtime, config, &tstate);
    if (_PyStatus_EXCEPTION(status)) {
        return status;
    }
    config = _PyInterpreterState_GetConfig(tstate->interp);

    if (config->_init_main) {
        // The main initialization phase
        status = pyinit_main(tstate);
        if (_PyStatus_EXCEPTION(status)) {
            return status;
        }
    }

    return _PyStatus_OK();
}
```

We can clearly see the separation between the initialization phases. The core phase is done by `pyinit_core()`, and the main phase is done by `pyinit_main()`. The `pyinit_core()` function initializes the "core" of Python. More specifically,

1. It prepares the configuration: parses the command line arguments, reads the environment variables, calculates the path configuration, chooses the encodings for the standard streams and the file system and writes all of this to the appropriate place in `config`.
2. It applies the configuration: configures the standard streams, generates the secret key for hashing, creates the main interpreter state and the main thread state, initializes the GIL and takes it, enables the GC, initializes built-in types and exceptions, initializes the `sys` module and the `builtins` module and sets up the import system for built-in and frozen modules.

During the first step, CPython calculates `config.module_search_paths`, which will be later copied to `sys.path`. Otherwise, this step is not very interesting, so let's look at `pyinit_config()` , which `pyinit_core()` calls to perform the second step:

```
static PyStatus
pyinit_config(_PyRuntimeState *runtime,
              PyThreadState **tstate_p,
              const PyConfig *config)
{
    // Set Py_* global variables from config.
    // Initialize C standard streams (stdin, stdout, stderr).
    // Set secret key for hashing.
    PyStatus status = pycore_init_runtime(runtime, config);
    if (_PyStatus_EXCEPTION(status)) {
        return status;
    }

    PyThreadState *tstate;
    // Create the main interpreter state and the main thread state.
    // Take the GIL.
    status = pycore_create_interpreter(runtime, config, &tstate);
    if (_PyStatus_EXCEPTION(status)) {
        return status;
```

```
    }
    *tstate_p = tstate;

    // Init types, exception, sys, builtins, importlib, etc.
    status = pycore_interp_init(tstate);
    if (_PyStatus_EXCEPTION(status)) {
        return status;
    }

    /* Only when we get here is the runtime core fully initialized */
    runtime->core_initialized = 1;
    return _PyStatus_OK();
}
```

First, `pycore_init_runtime()` copies some of the `config`'s fields to the corresponding global configuration variables. These global variables were used to configure CPython before `PyConfig` was introduced and continue to be a part of the Python/C API.

Next, `pycore_init_runtime()` sets the buffering modes for the `stdio`, `stdout` and `stderr` file pointers. On Unix-like systems this is done by calling the `setvbuf()` library function.

Finally, `pycore_init_runtime()` generates the secret key for hashing, which is stored in the `_Py_HashSecret` global variable. The secret key is taken along with the input by the SipHash24 hash function, which CPython uses to compute hashes. The secret key is randomly generated each time CPython starts. The purpose of randomization is to protect a Python application from hash collision DoS attacks. Python and many other languages including PHP, Ruby, JavaScript and C# were once vulnerable to such attacks. An attacker could send a set of strings with the same hash to an application and increase dramatically the CPU time required to put these strings in the dictionary because they all happen to be in the same bucket. The solution is to supply a hash function with the randomly generated key unknown to the attacker. Python also allows to generate a key deterministically by setting the `PYTHONHASHSEED` environment variable to some fixed value. To learn more about the attack, check this presentation. To learn more about the CPython's hash algorithm, check PEP 456.

In the first part we learned that CPython uses a thread state to store thread-specific data, such as a call stack and an exception state, and an interpreter state to store interpreter-specific data, such as loaded modules and import settings. The `pycore_create_interpreter()` function creates an interpreter state and a thread state for the main OS thread. We haven't seen what these structures look like yet, so here's the definition of the interpreter state struct:

```
// The PyInterpreterState typedef is in Include/pystate.h.
struct _is {

    // _PyRuntime.interpreters.head stores the most recently created interpreter
    // `next` allows to access all interpreters.
    struct _is *next;
    // `tstate_head` points to the most recently created thread state.
    // Thread states of the same interpreter are linked together.
    struct _ts *tstate_head;

    /* Reference to the _PyRuntime global variable. This field exists
       to not have to pass runtime in addition to tstate to a function.
       Get runtime from tstate: tstate->interp->runtime. */
    struct pyruntimestate *runtime;

    int64_t id;
    // For tracking references to the interpreter
    int64_t id_refcount;
    int requires_idref;
    PyThread_type_lock id_mutex;

    int finalizing;
```

```
    struct _ceval_state ceval;
    struct _gc_runtime_state gc;

    PyObject *modules;  // sys.modules points to it
    PyObject *modules_by_index;
    PyObject *sysdict;  // points to sys.__dict__
    PyObject *builtins; // points to builtins.__dict__
    PyObject *importlib;

    // A list of codec search functions
    PyObject *codec_search_path;
    PyObject *codec_search_cache;
    PyObject *codec_error_registry;
    int codecs_initialized;

    struct _Py_unicode_state unicode;

    PyConfig config;

    PyObject *dict;  /* Stores per-interpreter state */

    PyObject *builtins_copy;
    PyObject *import_func;
    /* Initialized to PyEval_EvalFrameDefault(). */
    _PyFrameEvalFunction eval_frame;

    // See `atexit` module
    void (*pyexitfunc)(PyObject *);
    PyObject *pyexitmodule;

    uint64_t tstate_next_unique_id;

    // See `warnings` module
    struct _warnings_runtime_state warnings;

    // A list of audit hooks, see sys.addaudithook
    PyObject *audit_hooks;

#if _PY_NSMALLNEGINTS + _PY_NSMALLPOSINTS > 0
    // Small integers are preallocated in this array so that they can be shared.
    // The default range is [-5, 256].
    PyLongObject* small_ints[_PY_NSMALLNEGINTS + _PY_NSMALLPOSINTS];
#endif

  // ... less interesting stuff for now
};
```

The important thing to note here is that `config` belongs to the interpreter state. The configuration that was read before is stored in `config` of the newly created interpreter state. The thread state struct is defined as follows:

```
// The PyThreadState typedef is in Include/pystate.h.
struct _ts {

    // Double-linked list is used to access all thread states belonging to the same interpreter
    struct _ts *prev;
    struct _ts *next;
    PyInterpreterState *interp;

    // Reference to the current frame (it can be NULL).
    // The call stack is accesible via frame->f_back.
    PyFrameObject *frame;

    // ... checking if recursion level is too deep
```

```
    // ... tracing/profiling

    /* The exception currently being raised */
    PyObject *curexc_type;
    PyObject *curexc_value;
    PyObject *curexc_traceback;

    /* The exception currently being handled, if no coroutines/generators
     * are present. Always last element on the stack referred to be exc_info.
     */
    _PyErr_StackItem exc_state;

    /* Pointer to the top of the stack of the exceptions currently
     * being handled */
    _PyErr_StackItem *exc_info;

    PyObject *dict;  /* Stores per-thread state */

    int gilstate_counter;

    PyObject *async_exc; /* Asynchronous exception to raise */
    unsigned long thread_id; /* Thread id where this tstate was created */

    /* Unique thread state id. */
    uint64_t id;

    // ... less interesting stuff for now
};
```

After creating a thread state for the main OS thread, `pycore_create_interpreter()` initializes the GIL, which prevents multiple threads from working with Python objects at the same time. If you spawn a new thread using the `threading` module, it starts executing a given target in the evaluation loop. In this case, threads wait for the GIL and take it at the start of each iteration of the evaluation loop. A thread can access its thread state because it's passed as an argument to the evaluation function. If you, however, write a C extension and call the Python/C API to take the GIL, CPython needs not only to take the GIL but also to associate the current thread with the corresponding thread state. This is done by storing a thread state in the thread specific storage (the `pthread_setspecific()` library function on Unix-like systems) upon the thread state creation. And that is the mechanism that allows any thread to access its thread state. The GIL deserves a separate post. The same is also true for the Python object system and the import mechanism, which we will also mention briefly in this post.

After the first interpreter state and the first thread state are created, `pyinit_config()` calls `pycore_interp_init()` to finish the core initialization phase. The code of `pycore_interp_init()` is self-explanatory:

```
static PyStatus
pycore_interp_init(PyThreadState *tstate)
{
    PyStatus status;
    PyObject *sysmod = NULL;

    status = pycore_init_types(tstate);
    if (_PyStatus_EXCEPTION(status)) {
        goto done;
    }

    status = _PySys_Create(tstate, &sysmod);
    if (_PyStatus_EXCEPTION(status)) {
        goto done;
    }

    status = pycore_init_builtins(tstate);
    if (_PyStatus_EXCEPTION(status)) {
        goto done;
```

```
        }

        status = pycore_init_import_warnings(tstate, sysmod);

    done:
        // Py_XDECREF() decreases the reference count of an object.
        // If the reference count becomes 0, the object is deallocated.
        Py_XDECREF(sysmod);
        return status;
    }
```

The `pycore_init_types()` function initializes built-in types. But what does it mean? And what are types really? As you probably know, everything you work with in Python is an object. Numbers, strings, lists, functions, modules, frame objects, user-defined classes and built-in types are all Python objects. A Python object is an instance of the `PyObject` struct or an instance of any other C struct whose first field is of type `PyObject`. The `PyObject` struct has two fields. The first field of type `Py_ssize_t` stores a reference count. The second field of type `PyTypeObject` points to the Python type of the object. Here's the definition of `PyObject`:

```
typedef struct _object {
    _PyObject_HEAD_EXTRA // for debugging only
    Py_ssize_t ob_refcnt;
    PyTypeObject *ob_type;
} PyObject;
```

And here's an example of a more familiar Python object, `float`:

```
typedef struct {
    PyObject_HEAD // macro that expands to PyObject ob_base;
    double ob_fval;
} PyFloatObject;
```

The C standard states that a pointer to any struct can be converted to a pointer to its first member and vice versa. Because any Python object has `PyObject` as its first member, CPython can treat any Python object as `PyObject`, which it does all over the place. You can think of it as of C way of doing subclassing. The benefit of such a trick is that it allows to achieve polymorphism. For example, it allows to write a function that can take any Python object as an argument by taking `PyObject`.

The reason why CPython can do something useful with `PyObject` is because the behavior of a Python object is determined by its type, and `PyObject` always has a type. A type "knows" how to create the objects of that type, how to calculate their hashes, how to add them, how to call them, how to access their attributes, how to deallocate them and much more. Types are also Python objects represented by the `PyTypeObject` structure. All types have the same type, which is `PyType_Type`. And the type of `PyType_Type` points to `PyType_Type` itself. If this explanation seems complicated, this example should not:

```
$ ./python.exe -q
>>> type([])
<class 'list'>
>>> type(type([]))
<class 'type'>
>>> type(type(type([])))
<class 'type'>
```

The fields of `PyTypeObject` are very well documented in the Python/C API Reference Manual. I only leave here the definition of the struct underlying `PyTypeObject` to get an idea of the amount of information that a Python type stores:

```
// PyTypeObject is a typedef for struct _typeobject
struct _typeobject {
```

```
    PyObject_VAR_HEAD // expands to
                      // PyObject ob_base;
                      // Py_ssize_t ob_size;
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                    or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */
    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;

    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    struct PyMethodDef *tp_methods;
    struct PyMemberDef *tp_members;
    struct PyGetSetDef *tp_getset;
    struct _typeobject *tp_base;
    PyObject *tp_dict;
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    Py_ssize_t tp_dictoffset;
    initproc tp_init;
    allocfunc tp_alloc;
```

```
    newfunc tp_new;
    freefunc tp_free; /* Low-level free-memory routine */
    inquiry tp_is_gc; /* For PyObject_IS_GC */
    PyObject *tp_bases;
    PyObject *tp_mro; /* method resolution order */
    PyObject *tp_cache;
    PyObject *tp_subclasses;
    PyObject *tp_weaklist;
    destructor tp_del;

    /* Type attribute cache version tag. Added in version 2.6 */
    unsigned int tp_version_tag;

    destructor tp_finalize;
    vectorcallfunc tp_vectorcall;
};
```

The built-in types, such as `int` and `list`, are implemented by statically defining instances of `PyTypeObject`, like so:

```
PyTypeObject PyList_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "list",
    sizeof(PyListObject),
    0,
    (destructor)list_dealloc,                   /* tp_dealloc */
    0,                                          /* tp_vectorcall_offset */
    0,                                          /* tp_getattr */
    0,                                          /* tp_setattr */
    0,                                          /* tp_as_async */
    (reprfunc)list_repr,                        /* tp_repr */
    0,                                          /* tp_as_number */
    &list_as_sequence,                          /* tp_as_sequence */
    &list_as_mapping,                           /* tp_as_mapping */
    PyObject_HashNotImplemented,                /* tp_hash */
    0,                                          /* tp_call */
    0,                                          /* tp_str */
    PyObject_GenericGetAttr,                    /* tp_getattro */
    0,                                          /* tp_setattro */
    0,                                          /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_HAVE_GC |
        Py_TPFLAGS_BASETYPE | Py_TPFLAGS_LIST_SUBCLASS, /* tp_flags */
    list___init____doc__,                       /* tp_doc */
    (traverseproc)list_traverse,                /* tp_traverse */
    (inquiry)_list_clear,                       /* tp_clear */
    list_richcompare,                           /* tp_richcompare */
    0,                                          /* tp_weaklistoffset */
    list_iter,                                  /* tp_iter */
    0,                                          /* tp_iternext */
    list_methods,                               /* tp_methods */
    0,                                          /* tp_members */
    0,                                          /* tp_getset */
    0,                                          /* tp_base */
    0,                                          /* tp_dict */
    0,                                          /* tp_descr_get */
    0,                                          /* tp_descr_set */
    0,                                          /* tp_dictoffset */
    (initproc)list___init__,                    /* tp_init */
    PyType_GenericAlloc,                        /* tp_alloc */
    PyType_GenericNew,                          /* tp_new */
    PyObject_GC_Del,                            /* tp_free */
    .tp_vectorcall = list_vectorcall,
};
```

CPython needs also to initialize the built-in types. This is what we started our discussion of types with. All types require some initialization, for example, to add special methods, like `__call__` and `__eq__`, to the type's dictionary and point them to the corresponding `tp_*` functions. This common initialization is done by calling `PyType_Ready()` for each type:

```
PyStatus
_PyTypes_Init(void)
{
    // The names of the special methods "__hash__", "__call_", etc. are interned by this call
    PyStatus status = _PyTypes_InitSlotDefs();
    if (_PyStatus_EXCEPTION(status)) {
        return status;
    }

#define INIT_TYPE(TYPE, NAME) \
    do { \
        if (PyType_Ready(TYPE) < 0) { \
            return _PyStatus_ERR("Can't initialize " NAME " type"); \
        } \
    } while (0)

    INIT_TYPE(&PyBaseObject_Type, "object");
    INIT_TYPE(&PyType_Type, "type");
    INIT_TYPE(&_PyWeakref_RefType, "weakref");
    INIT_TYPE(&_PyWeakref_CallableProxyType, "callable weakref proxy");
    INIT_TYPE(&_PyWeakref_ProxyType, "weakref proxy");
    INIT_TYPE(&PyLong_Type, "int");
    INIT_TYPE(&PyBool_Type, "bool");
    INIT_TYPE(&PyByteArray_Type, "bytearray");
    INIT_TYPE(&PyBytes_Type, "str");
    INIT_TYPE(&PyList_Type, "list");
    INIT_TYPE(&_PyNone_Type, "None");
    INIT_TYPE(&_PyNotImplemented_Type, "NotImplemented");
    INIT_TYPE(&PyTraceBack_Type, "traceback");
    INIT_TYPE(&PySuper_Type, "super");
    INIT_TYPE(&PyRange_Type, "range");
    INIT_TYPE(&PyDict_Type, "dict");
    INIT_TYPE(&PyDictKeys_Type, "dict keys");
    // ... 50 more types
    return _PyStatus_OK();

#undef INIT_TYPE
}
```

Some built-in types require additional type-specific initialization. For example, the initialization of `int` is needed to preallocate small integers in the `interp->small_ints` array so that they can be reused, and the initialization of `float` is needed to determine how the current machine represents the floating point number.

When the built-in types are initialized, `pycore_interp_init()` calls `_PySys_Create()` to create the `sys` module. Why is the `sys` module is the first module to be created? Of course, it's very important, since it contains such things as the command line arguments passed to a program (`sys.argv`), the list of path entries to search for modules (`sys.path`), a lot of system-specific and implementation-specific data (`sys.version`, `sys.implementation`, `sys.thread_info`, etc.) and various functions that allow to interact with the interpreter (`sys.addaudithook()`, `sys.settrace()`, etc.). The main reason, though, to create the `sys` module so early is to initialize `sys.modules`. It points to the `interp->modules` dictionary, which is also created by `_PySys_Create()`, and acts as a cache for all imported modules. It's the first place to look up for a module, and it's the place where all loaded modules are saved to. The import system heavily relies on `sys.modules`.

After the call to `_PySys_Create()`, the `sys` module is only partially initialized. The functions and most of the variables are available, but invocation-specific data, such as `sys.argv` and `sys._xoptions`, and the path-related configuration, such as `sys.path` and `sys.exec_prefix`, will be set during the main initialization phase.

When the `sys` module is created, `pycore_interp_init()` calls `pycore_init_builtins()` to initialize the `builtins` module. The built-in functions, like `abs()`, `dir()` and `print()`, the built-in types, like `dict`, `int` and `str`, the built-in exceptions, like `Exception` and `ValueError`, and the built-in constants, like `False`, `Ellipsis` and `None`, are all members of the `builtins` module. The built-in functions are a part of the module definition, but other members have to be placed in the module's dictionary explicitly. The `pycore_init_builtins()` function does that. Later on, `frame->f_builtins` will be set to this dictionary to look up names. This is why we don't need to import `builtins` directly.

The last step of the core initialization phase is performed by the `pycore_init_import_warnings()` function. You probably know that Python has a mechanism to issue warnings, like so:

```
$ ./python.exe -q
>>> import imp
<stdin>:1: DeprecationWarning: the imp module is deprecated in favour of importlib; ...
```

Warnings can be ignored, turned into exceptions and displayed in various ways. CPython has filters to do that. Some filters are turned on by default, and the `pycore_init_import_warnings()` function is what turns them on. Most crucially, though, is that `pycore_init_import_warnings()` sets up the import system for built-in and frozen modules.

The built-in and frozen modules are two special kinds of modules. What unites them is that they are compiled directly into the `python` executable. The difference is that built-in modules are written in C, while frozen modules are written in Python. How is that possible to compile a module written in Python into the executable? This is cleverly done by incorporating the binary representation of a module's code object into the C source code. To generate the binary representation, the Freeze utility is used.

An example of a frozen module is `_frozen_importlib`. This is the core of the import system. Python's `import` statement eventually leads to the `_frozen_importlib._find_and_load()` function. To support the import of the built-in and frozen modules, `pycore_init_import_warnings()` calls `init_importlib()`, and the very first thing `init_importlib()` does is import `_frozen_importlib`. It may seem that CPython has to import `_frozen_importlib` in order to import `_frozen_importlib`, but this is not the case. The `_frozen_importlib` module is a part of the universal API for importing any module. If CPython, however, knows that it needs to import a frozen module, it can do so without reliance on `_frozen_importlib`.

The `_frozen_importlib` module depends on two other modules. First, it needs the `sys` module to get an access to `sys.modules`. Second, it needs the `_imp` module, which implements low-level import functions, including the functions for creating built-in and frozen modules. The problem is that `_frozen_importlib` cannot import any modules because the `import` statement depends on `_frozen_importlib` itself. The solution is to create the `_imp` module in `init_importlib()` and inject it and the `sys` module in `_frozen_importlib` by calling `_frozen_importlib._install(sys, _imp)`. This bootstrapping of the import system ends the core initialization phase.

We leave `pyinit_core()` and enter `pyinit_main()`, which performs the main initialization phase. We find that it does some checks and calls `init_interp_main()` to do the work. The work can be summarized as follows:

1. Get system's realtime and monotonic clocks, ensure that `time.time()`, `time.monotonic()` and `time.perf_counter()` will work correctly.

2. Finish the initialization of the `sys` module. This includes setting the path configuration variables, such as `sys.path`, `sys.executable` and `sys.exec_prefix`, and invocation-specific variables, such as `sys.argv` and `sys._xoptions`.

3. Add support for the import of path-based (external) modules. This is done by importing another frozen module called `importlib._bootstrap_external`. It enables the import of modules based on `sys.path`. Also, the `zipimport` frozen module is imported. It enables the import of modules from ZIP archives, i.e. the directories listed in the `sys.path` can be ZIP archives.

4. Normalize the names of the encodings for the file system and the standard streams. Set the error handlers for encoding and decoding when dealing with the file system.

5. Install default signal handlers. These are the handlers that get executed when a process receives a signal like `SIGINT`. The custom handlers can be set up using the `signal` module.

6. Import the `io` module and initialize `sys.stdin`, `sys.stdout` and `sys.stderr`. This is essentially done by calling `io.open()` on the file descriptors for the standard streams.

7. Set `builtins.open` to `io.OpenWrapper`, so that `open()` is available as a built-in function.

8. Create the `__main__` module, set `__main__.__builtins__` to `builtins` and `__main__.__loader__` to `_frozen_importlib.BuiltinImporter`. At this point, the `__main__` module contains nothing else.

9. Import `warnings` and `site` modules. The `site` module adds the site-specific directories to `sys.path`. This is why `sys.path` normally contains a directory with the installed modules like `/usr/local/lib/python3.9/site-packages/`.

10. Set `interp->runtime->initialized = 1`

The initialization of CPython is completed. The `pymain_init()` function returns, and we step into `Py_RunMain()` to see what else CPython does before it enters the evaluation loop.

running a Python program

The `Py_RunMain()` function doesn't seem like a place where the action happens:

```
int
Py_RunMain(void)
{
    int exitcode = 0;

    pymain_run_python(&exitcode);

    if (Py_FinalizeEx() < 0) {
        /* Value unlikely to be confused with a non-error exit status or
           other special meaning */
        exitcode = 120;
    }

    // Free the memory that is not freed by Py_FinalizeEx()
    pymain_free();

    if (_Py_UnhandledKeyboardInterrupt) {
        exitcode = exit_sigint();
    }

    return exitcode;
}
```

First, `Py_RunMain()` calls `pymain_run_python()` to run Python. Second, it calls `Py_FinalizeEx()` to undo the initialization. The `Py_FinalizeEx()` functions frees most of the memory that CPython is able to free, and the rest is freed by `pymain_free()`. Another important reason to finalize CPython is to call the exit functions, including the functions registered with the `atexit` module.

As you probably know, there are number of ways to run `python`, namely:

- interactively:

```
$ ./cpython/python.exe
>>> import sys
>>> sys.path[:1]
['']
```

- from stdin:

```
$ echo "import sys; print(sys.path[:1])" | ./cpython/python.exe
['']
```

- as a command:

```
$ ./cpython/python.exe -c "import sys; print(sys.path[:1])"
['']
```

- as a script

```
$ ./cpython/python.exe 03/print_path0.py
['/Users/Victor/Projects/tenthousandmeters/python_behind_the_scenes/03']
```

- as a module:

```
$ ./cpython/python.exe -m 03.print_path0
['/Users/Victor/Projects/tenthousandmeters/python_behind_the_scenes']
```

- and, less obvious, package as a script (`print_path0_package` is a directory with `__main__.py`):

```
$ ./cpython/python.exe 03/print_path0_package
['/Users/Victor/Projects/tenthousandmeters/python_behind_the_scenes/03/print_path0_package']
```

I moved one level up from the `cpython/` directory to show that different modes of invocation lead to different values of `sys.path[0]`. What the next function on our way, `pymain_run_python()`, does is compute the value of `sys.path[0]`, prepend it to `sys.path` and run Python in the appropriate mode according to `config`:

```
static void
pymain_run_python(int *exitcode)
{
    PyInterpreterState *interp = _PyInterpreterState_GET();
    PyConfig *config = (PyConfig*)_PyInterpreterState_GetConfig(interp);

    // Prepend the search path to `sys.path`
    PyObject *main_importer_path = NULL;
    if (config->run_filename != NULL) {
        // Calculate the search path for the case when the filename is a package
        // (ex: directory or ZIP file) which contains __main__.py, store it in `main_importer_path`.
        // Otherwise, left `main_importer_path` unchanged.
        // Handle other cases later.
        if (pymain_get_importer(config->run_filename, &main_importer_path,
                                exitcode)) {
            return;
        }
    }

    if (main_importer_path != NULL) {
        if (pymain_sys_path_add_path0(interp, main_importer_path) < 0) {
            goto error;
        }
    }
    else if (!config->isolated) {
        PyObject *path0 = NULL;
        // Compute the search path that will be prepended to `sys.path` for other cases.
        // If running as script, then it's the directory where the script is located.
        // If running as module (-m), then it's the current working directory.
        // Otherwise, it's an empty string.
```

```
            int res = _PyPathConfig_ComputeSysPath0(&config->argv, &path0);
            if (res < 0) {
                goto error;
            }

            if (res > 0) {
                if (pymain_sys_path_add_path0(interp, path0) < 0) {
                    Py_DECREF(path0);
                    goto error;
                }
                Py_DECREF(path0);
            }
        }

        PyCompilerFlags cf = _PyCompilerFlags_INIT;

        // Print version and platform in the interactive mode
        pymain_header(config);
        // Import `readline` module to provide completion,
        // line editing and history capabilities in the interactive mode
        pymain_import_readline(config);

        // Run Python depending on the mode of invocation (script, -m, -c, etc.)
        if (config->run_command) {
            *exitcode = pymain_run_command(config->run_command, &cf);
        }
        else if (config->run_module) {
            *exitcode = pymain_run_module(config->run_module, 1);
        }
        else if (main_importer_path != NULL) {
            *exitcode = pymain_run_module(L"__main__", 0);
        }
        else if (config->run_filename != NULL) {
            *exitcode = pymain_run_file(config, &cf);
        }
        else {
            *exitcode = pymain_run_stdin(config, &cf);
        }

        // Enter the interactive mode after executing a program.
        // Enabled by `-i` and `PYTHONINSPECT`.
        pymain_repl(config, &cf, exitcode);
        goto done;

    error:
        *exitcode = pymain_exit_err_print();

    done:
        Py_XDECREF(main_importer_path);
    }
```

We won't follow all paths, but assume that we run a Python program as a script. This leads us to the `pymain_run_file()` function, which checks whether the specified file can be opened, ensures it's not a directory and calls `PyRun_AnyFileExFlags()`. The `PyRun_AnyFileExFlags()` function handles a special case when the file is a terminal (`isatty(fd)` returns 1). If this is the case, it enters the interactive mode:

```
$ ./python.exe /dev/ttys000
>>> 1 + 1
2
```

Otherwise, it calls `PyRun_SimpleFileExFlags()`. You should be familiar with `.pyc` files that constantly pop up in the `__pycache__` directories alongside your modules. A `.pyc` file contains the compiled source code of a Python program, that

is, the marshaled code object of a module. Due to `.pyc` files, we don't need to recompile modules every time we import them. I guess you know that, but did you know that it's possible to run a `.pyc` file directly?

```
$ ./cpython/python.exe 03/__pycache__/print_path0.cpython-39.pyc
['/Users/Victor/Projects/tenthousandmeters/python_behind_the_scenes/03/__pycache__']
```

The `PyRun_SimpleFileExFlags()` function implements this logic. It checks whether the file is a `.pyc` file, whether it's compiled for the current CPython version and, if yes, calls `run_pyc_file()`. If the file is not a `.pyc` file, it calls `PyRun_FileExFlags()`. Most importantly, though, is that `PyRun_SimpleFileExFlags()` imports the `__main__` module and passes `__main__`'s dictionary to `PyRun_FileExFlags()` as the global and the local namespace in which the file will be executed:

```
int
PyRun_SimpleFileExFlags(FILE *fp, const char *filename, int closeit,
                        PyCompilerFlags *flags)
{
    PyObject *m, *d, *v;
    const char *ext;
    int set_file_name = 0, ret = -1;
    size_t len;

    m = PyImport_AddModule("__main__");
    if (m == NULL)
        return -1;
    Py_INCREF(m);
    d = PyModule_GetDict(m);

    if (PyDict_GetItemString(d, "__file__") == NULL) {
        PyObject *f;
        f = PyUnicode_DecodeFSDefault(filename);
        if (f == NULL)
            goto done;
        if (PyDict_SetItemString(d, "__file__", f) < 0) {
            Py_DECREF(f);
            goto done;
        }
        if (PyDict_SetItemString(d, "__cached__", Py_None) < 0) {
            Py_DECREF(f);
            goto done;
        }
        set_file_name = 1;
        Py_DECREF(f);
    }

    // Check if a .pyc file is passed
    len = strlen(filename);
    ext = filename + len - (len > 4 ? 4 : 0);
    if (maybe_pyc_file(fp, filename, ext, closeit)) {
        FILE *pyc_fp;
        /* Try to run a pyc file. First, re-open in binary */
        if (closeit)
            fclose(fp);
        if ((pyc_fp = _Py_fopen(filename, "rb")) == NULL) {
            fprintf(stderr, "python: Can't reopen .pyc file\n");
            goto done;
        }

        if (set_main_loader(d, filename, "SourcelessFileLoader") < 0) {
            fprintf(stderr, "python: failed to set __main__.__loader__\n");
            ret = -1;
            fclose(pyc_fp);
            goto done;
        }
        v = run_pyc_file(pyc_fp, filename, d, d, flags);
```

```
    } else {
        /* When running from stdin, leave __main__.__loader__ alone */
        if (strcmp(filename, "<stdin>") != 0 &&
            set_main_loader(d, filename, "SourceFileLoader") < 0) {
            fprintf(stderr, "python: failed to set __main__.__loader__\n");
            ret = -1;
            goto done;
        }
        v = PyRun_FileExFlags(fp, filename, Py_file_input, d, d,
                              closeit, flags);
    }
    flush_io();
    if (v == NULL) {
        Py_CLEAR(m);
        PyErr_Print();
        goto done;
    }
    Py_DECREF(v);
    ret = 0;
  done:
    if (set_file_name) {
        if (PyDict_DelItemString(d, "__file__")) {
            PyErr_Clear();
        }
        if (PyDict_DelItemString(d, "__cached__")) {
            PyErr_Clear();
        }
    }
    Py_XDECREF(m);
    return ret;
}
```

The `PyRun_FileExFlags()` function begins the compilation process. It runs the parser, gets back the AST of the module and calls `run_mod()` to run the AST. It also creates a `PyArena` object, which CPython uses to allocate small objects (smaller or equal to 512 bytes):

```
PyObject *
PyRun_FileExFlags(FILE *fp, const char *filename_str, int start, PyObject *globals,
                  PyObject *locals, int closeit, PyCompilerFlags *flags)
{
    PyObject *ret = NULL;
    mod_ty mod;
    PyArena *arena = NULL;
    PyObject *filename;
    int use_peg = _PyInterpreterState_GET()->config._use_peg_parser;

    filename = PyUnicode_DecodeFSDefault(filename_str);
    if (filename == NULL)
        goto exit;

    arena = PyArena_New();
    if (arena == NULL)
        goto exit;

    // Run the parser.
    // By default the new PEG parser is used.
    // Pass `-X oldparser` to use the old parser.
    // `mod` stands for module. It's the root node of the AST.
    if (use_peg) {
        mod = PyPegen_ASTFromFileObject(fp, filename, start, NULL, NULL, NULL,
                                        flags, NULL, arena);
    }
    else {
        mod = PyParser_ASTFromFileObject(fp, filename, NULL, start, 0, 0,
```

```
                                      flags, NULL, arena);
    }

    if (closeit)
        fclose(fp);
    if (mod == NULL) {
        goto exit;
    }
    // Compile the AST and run.
    ret = run_mod(mod, filename, globals, locals, flags, arena);

exit:
    Py_XDECREF(filename);
    if (arena != NULL)
        PyArena_Free(arena);
    return ret;
}
```

`run_mod()` runs the compiler by calling `PyAST_CompileObject()`, gets back the module's code object and calls `run_eval_code_obj()` to execute the code object. In the interim, it raises the `exec` event, which is a CPython's way to notify auditing tools when something important happens inside the Python runtime. PEP 578 explains this mechanism.

```
static PyObject *
run_mod(mod_ty mod, PyObject *filename, PyObject *globals, PyObject *locals,
            PyCompilerFlags *flags, PyArena *arena)
{
    PyThreadState *tstate = _PyThreadState_GET();
    PyCodeObject *co = PyAST_CompileObject(mod, filename, flags, -1, arena);
    if (co == NULL)
        return NULL;

    if (_PySys_Audit(tstate, "exec", "O", co) < 0) {
        Py_DECREF(co);
        return NULL;
    }

    PyObject *v = run_eval_code_obj(tstate, co, globals, locals);
    Py_DECREF(co);
    return v;
}
```

We already know from the second part that the compiler works by:

1. building a symbol table
2. creating a CFG of basic blocks; and
3. assembling the CFG into a code object.

This is exactly what `PyAST_CompileObject()` does, so we won't focus on it now.

`run_eval_code_obj()` begins a chain of trivial function calls that eventually leads us to `_PyEval_EvalCode()`. I paste all those functions here, just so you can see where the parameters of `_PyEval_EvalCode()` come from:

```
static PyObject *
run_eval_code_obj(PyThreadState *tstate, PyCodeObject *co, PyObject *globals, PyObject *locals)
{
    PyObject *v;
    // The special case when CPython is embeddded. We can safely ignore it.
    /*
     * We explicitly re-initialize _Py_UnhandledKeyboardInterrupt every eval
     * _just in case_ someone is calling into an embedded Python where they
```

```
         * don't care about an uncaught KeyboardInterrupt exception (why didn't they
         * leave config.install_signal_handlers set to 0?!?) but then later call
         * Py_Main() itself (which _checks_ this flag and dies with a signal after
         * its interpreter exits).  We don't want a previous embedded interpreter's
         * uncaught exception to trigger an unexplained signal exit from a future
         * Py_Main() based one.
         */
        _Py_UnhandledKeyboardInterrupt = 0;

        /* Set globals['__builtins__'] if it doesn't exist */
        // In our case, it's been already set to the `builtins` module during the main initialization.
        if (globals != NULL && PyDict_GetItemString(globals, "__builtins__") == NULL) {
            if (PyDict_SetItemString(globals, "__builtins__",
                                     tstate->interp->builtins) < 0) {
                return NULL;
            }
        }

        v = PyEval_EvalCode((PyObject*)co, globals, locals);
        if (!v && _PyErr_Occurred(tstate) == PyExc_KeyboardInterrupt) {
            _Py_UnhandledKeyboardInterrupt = 1;
        }
        return v;
    }


    PyObject *
    PyEval_EvalCode(PyObject *co, PyObject *globals, PyObject *locals)
    {
        return PyEval_EvalCodeEx(co,
                         globals, locals,
                         (PyObject **)NULL, 0,
                         (PyObject **)NULL, 0,
                         (PyObject **)NULL, 0,
                         NULL, NULL);
    }


    PyObject *
    PyEval_EvalCodeEx(PyObject *_co, PyObject *globals, PyObject *locals,
                      PyObject *const *args, int argcount,
                      PyObject *const *kws, int kwcount,
                      PyObject *const *defs, int defcount,
                      PyObject *kwdefs, PyObject *closure)
    {
        return _PyEval_EvalCodeWithName(_co, globals, locals,
                                        args, argcount,
                                        kws, kws != NULL ? kws + 1 : NULL,
                                        kwcount, 2,
                                        defs, defcount,
                                        kwdefs, closure,
                                        NULL, NULL);
    }


    PyObject *
    _PyEval_EvalCodeWithName(PyObject *_co, PyObject *globals, PyObject *locals,
               PyObject *const *args, Py_ssize_t argcount,
               PyObject *const *kwnames, PyObject *const *kwargs,
               Py_ssize_t kwcount, int kwstep,
               PyObject *const *defs, Py_ssize_t defcount,
               PyObject *kwdefs, PyObject *closure,
               PyObject *name, PyObject *qualname)
    {
        PyThreadState *tstate = _PyThreadState_GET();
        return _PyEval_EvalCode(tstate, _co, globals, locals,
```

```
                args, argcount,
                kwnames, kwargs,
                kwcount, kwstep,
                defs, defcount,
                kwdefs, closure,
                name, qualname);
}
```

Recall that a code object describes what a piece of code does, but to execute a code object, CPython needs to create a state for it, which is what a frame object is. `_PyEval_EvalCode()` creates a frame object for a given code object with specified parameters. In our case, most of the parameters are `NULL`, so little has to be done. Much more work is required when CPython executes, for example, a function's code object with different kinds of arguments passed. As a result, `_PyEval_EvalCode()` is nearly 300 lines long. We'll see what most of them are for in the next parts. For now, you can skip through `_PyEval_EvalCode()` to ensure that in the end it calls `_PyEval_EvalFrame()` to evaluate the created frame object:

```
PyObject *
_PyEval_EvalCode(PyThreadState *tstate,
           PyObject *_co, PyObject *globals, PyObject *locals,
           PyObject *const *args, Py_ssize_t argcount,
           PyObject *const *kwnames, PyObject *const *kwargs,
           Py_ssize_t kwcount, int kwstep,
           PyObject *const *defs, Py_ssize_t defcount,
           PyObject *kwdefs, PyObject *closure,
           PyObject *name, PyObject *qualname)
{
    assert(is_tstate_valid(tstate));

    PyCodeObject* co = (PyCodeObject*)_co;
    PyFrameObject *f;
    PyObject *retval = NULL;
    PyObject **fastlocals, **freevars;
    PyObject *x, *u;
    const Py_ssize_t total_args = co->co_argcount + co->co_kwonlyargcount;
    Py_ssize_t i, j, n;
    PyObject *kwdict;

    if (globals == NULL) {
        _PyErr_SetString(tstate, PyExc_SystemError,
                         "PyEval_EvalCodeEx: NULL globals");
        return NULL;
    }

    /* Create the frame */
    f = _PyFrame_New_NoTrack(tstate, co, globals, locals);
    if (f == NULL) {
        return NULL;
    }
    fastlocals = f->f_localsplus;
    freevars = f->f_localsplus + co->co_nlocals;

    /* Create a dictionary for keyword parameters (**kwags) */
    if (co->co_flags & CO_VARKEYWORDS) {
        kwdict = PyDict_New();
        if (kwdict == NULL)
            goto fail;
        i = total_args;
        if (co->co_flags & CO_VARARGS) {
            i++;
        }
        SETLOCAL(i, kwdict);
    }
    else {
        kwdict = NULL;
```

```
    }

    /* Copy all positional arguments into local variables */
    if (argcount > co->co_argcount) {
        n = co->co_argcount;
    }
    else {
        n = argcount;
    }
    for (j = 0; j < n; j++) {
        x = args[j];
        Py_INCREF(x);
        SETLOCAL(j, x);
    }

    /* Pack other positional arguments into the *args argument */
    if (co->co_flags & CO_VARARGS) {
        u = _PyTuple_FromArray(args + n, argcount - n);
        if (u == NULL) {
            goto fail;
        }
        SETLOCAL(total_args, u);
    }

    /* Handle keyword arguments passed as two strided arrays */
    kwcount *= kwstep;
    for (i = 0; i < kwcount; i += kwstep) {
        PyObject **co_varnames;
        PyObject *keyword = kwnames[i];
        PyObject *value = kwargs[i];
        Py_ssize_t j;

        if (keyword == NULL || !PyUnicode_Check(keyword)) {
            _PyErr_Format(tstate, PyExc_TypeError,
                          "%U() keywords must be strings",
                          co->co_name);
            goto fail;
        }

        /* Speed hack: do raw pointer compares. As names are
           normally interned this should almost always hit. */
        co_varnames = ((PyTupleObject *)(co->co_varnames))->ob_item;
        for (j = co->co_posonlyargcount; j < total_args; j++) {
            PyObject *name = co_varnames[j];
            if (name == keyword) {
                goto kw_found;
            }
        }

        /* Slow fallback, just in case */
        for (j = co->co_posonlyargcount; j < total_args; j++) {
            PyObject *name = co_varnames[j];
            int cmp = PyObject_RichCompareBool( keyword, name, Py_EQ);
            if (cmp > 0) {
                goto kw_found;
            }
            else if (cmp < 0) {
                goto fail;
            }
        }

        assert(j >= total_args);
        if (kwdict == NULL) {

            if (co->co_posonlyargcount
                && positional_only_passed_as_keyword(tstate, co,
```

```
                                                kwcount, kwnames))
            {
                goto fail;
            }

            _PyErr_Format(tstate, PyExc_TypeError,
                          "%U() got an unexpected keyword argument '%S'",
                          co->co_name, keyword);
            goto fail;
        }

        if (PyDict_SetItem(kwdict, keyword, value) == -1) {
            goto fail;
        }
        continue;

      kw_found:
        if (GETLOCAL(j) != NULL) {
            _PyErr_Format(tstate, PyExc_TypeError,
                          "%U() got multiple values for argument '%S'",
                          co->co_name, keyword);
            goto fail;
        }
        Py_INCREF(value);
        SETLOCAL(j, value);
    }

    /* Check the number of positional arguments */
    if ((argcount > co->co_argcount) && !(co->co_flags & CO_VARARGS)) {
        too_many_positional(tstate, co, argcount, defcount, fastlocals);
        goto fail;
    }

    /* Add missing positional arguments (copy default values from defs) */
    if (argcount < co->co_argcount) {
        Py_ssize_t m = co->co_argcount - defcount;
        Py_ssize_t missing = 0;
        for (i = argcount; i < m; i++) {
            if (GETLOCAL(i) == NULL) {
                missing++;
            }
        }
        if (missing) {
            missing_arguments(tstate, co, missing, defcount, fastlocals);
            goto fail;
        }
        if (n > m)
            i = n - m;
        else
            i = 0;
        for (; i < defcount; i++) {
            if (GETLOCAL(m+i) == NULL) {
                PyObject *def = defs[i];
                Py_INCREF(def);
                SETLOCAL(m+i, def);
            }
        }
    }

    /* Add missing keyword arguments (copy default values from kwdefs) */
    if (co->co_kwonlyargcount > 0) {
        Py_ssize_t missing = 0;
        for (i = co->co_argcount; i < total_args; i++) {
            PyObject *name;
            if (GETLOCAL(i) != NULL)
                continue;
```

```
            name = PyTuple_GET_ITEM(co->co_varnames, i);
            if (kwdefs != NULL) {
                PyObject *def = PyDict_GetItemWithError(kwdefs, name);
                if (def) {
                    Py_INCREF(def);
                    SETLOCAL(i, def);
                    continue;
                }
                else if (_PyErr_Occurred(tstate)) {
                    goto fail;
                }
            }
            missing++;
        }
        if (missing) {
            missing_arguments(tstate, co, missing, -1, fastlocals);
            goto fail;
        }
    }

    /* Allocate and initialize storage for cell vars, and copy free
       vars into frame. */
    for (i = 0; i < PyTuple_GET_SIZE(co->co_cellvars); ++i) {
        PyObject *c;
        Py_ssize_t arg;
        /* Possibly account for the cell variable being an argument. */
        if (co->co_cell2arg != NULL &&
            (arg = co->co_cell2arg[i]) != CO_CELL_NOT_AN_ARG) {
            c = PyCell_New(GETLOCAL(arg));
            /* Clear the local copy. */
            SETLOCAL(arg, NULL);
        }
        else {
            c = PyCell_New(NULL);
        }
        if (c == NULL)
            goto fail;
        SETLOCAL(co->co_nlocals + i, c);
    }

    /* Copy closure variables to free variables */
    for (i = 0; i < PyTuple_GET_SIZE(co->co_freevars); ++i) {
        PyObject *o = PyTuple_GET_ITEM(closure, i);
        Py_INCREF(o);
        freevars[PyTuple_GET_SIZE(co->co_cellvars) + i] = o;
    }

    /* Handle generator/coroutine/asynchronous generator */
    if (co->co_flags & (CO_GENERATOR | CO_COROUTINE | CO_ASYNC_GENERATOR)) {
        PyObject *gen;
        int is_coro = co->co_flags & CO_COROUTINE;

        /* Don't need to keep the reference to f_back, it will be set
         * when the generator is resumed. */
        Py_CLEAR(f->f_back);

        /* Create a new generator that owns the ready to run frame
         * and return that as the value. */
        if (is_coro) {
            gen = PyCoro_New(f, name, qualname);
        } else if (co->co_flags & CO_ASYNC_GENERATOR) {
            gen = PyAsyncGen_New(f, name, qualname);
        } else {
            gen = PyGen_NewWithQualName(f, name, qualname);
        }
        if (gen == NULL) {
```

```
            return NULL;
        }

        _PyObject_GC_TRACK(f);

        return gen;
    }

    retval = _PyEval_EvalFrame(tstate, f, 0);

fail: /* Jump here from prelude on failure */

    /* decref'ing the frame can cause __del__ methods to get invoked,
       which can call back into Python.  While we're done with the
       current Python frame (f), the associated C stack is still in use,
       so recursion_depth must be boosted for the duration.
    */
    if (Py_REFCNT(f) > 1) {
        Py_DECREF(f);
        _PyObject_GC_TRACK(f);
    }
    else {
        ++tstate->recursion_depth;
        Py_DECREF(f);
        --tstate->recursion_depth;
    }
    return retval;
}
```

`_PyEval_EvalFrame()` is a wrapper around `interp->eval_frame()`, which is the frame evaluation function. It's possible to set `interp->eval_frame()` to a custom function. Why would someone want to do that? For example, it allows to add a JIT compiler to CPython by replacing the default evaluation function with the one that stores compiled machine code in a code object and runs it. PEP 523 introduced this functionality in CPython 3.6.

By default, `interp->eval_frame()` is set to `_PyEval_EvalFrameDefault()`. This function, defined in `Python/ceval.c`, consists of almost 3,000 lines. Today, though, we're only interested in one. Line 1336 of `Python/ceval.c` begins what we've been waiting for so long: the evaluation loop.

Conclusion

We've discussed a lot today. We started by making an overview of the CPython project, compiled CPython and stepped through its source code, studying the initialization stage along the way. Overall, I think this should give you a descent understanding of what CPython does before it starts interpreting the bytecode. What happens after is the subject of the next post.

Meanwhile, to solidify what we learned today and to learn more, I really recommend you to find some time to explore the CPython source code on your own. I bet you have many questions after reading this post, so you should have something to look for. Have a good time!

If you have any questions, comments or suggestions, feel free to contact me at victor@tenthousandmeters.com

Proudly powered by Pelican, which takes great advantage of Python.

The theme is by Smashing Magazine, thanks!