Internals of Python 3.x

Derive Maximum Code Performance and Delve Further into Iterations, Objects, GIL, Memory management, and various Internals

PRASHANTH RAGHU

bpb

Internals of Python 3.x

Derive Maximum Code Performance and Delve Further into Iterations, Objects, GIL, Memory Management, and Various Internals

Prashanth Raghu

www.bpbonline.com

FIRST EDITION 2022

ISBN: 978-93-91030-940

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

Dedicated to

My Grandparents
Late Sri. G Sampath and Late G.S. Srirangamma

Late Sri. S.R. Srinivasa Iyengar and
Smt Jayalakshmi Srinivasa Iyengar

My beloved Parents
Sri. S. Raghu and Smt. S. Anuradha
Brother - Jayanth R

Friends, Family and Well Wishers

Special dedication and credits to revered Guido Van Rossum and the team of contributors to CPython. This book is just an effort to summarize your hard work and dedication and to serve as an unofficial documentation to the source to help developers make better decisions.

About the Author

Prashanth graduated with Masters from National University of Singapore and Bachelors from PES Institute of Technology, Bangalore and has keen interest towards contributing to open source technologies. His interests lie in learning new technologies and their impact on developing scalable solutions. He currently works as a Staff Software Engineer at InMobi Technologies aiming to develop solutions at scale for the advertising domain. He has been pursuing Carnatic music initially under the tutelage of Sri. Vid. S Venkatesh and currently under the guidance of Sri. Vid. H.S. Venugopal.

About the Reviewer

Rajat Joginder Singh has more than 5 years of work experience in the software industry, explicitly in the cloud application development domain using Python. He is currently working as a Backend Developer at SignEasy. He has worked on Backend Development using frameworks such as Flask, Django and has knowledge on Dockerized applications.

Acknowledgement

This book would not have been possible without the infinite support of my family. First and foremost, my grandfather Mr. G. Sampath a leading advocate in Bangalore, India who led the baton in the Central Excise and Customs department to become one of the authorities on the subject matter in India. I would like to acknowledge the infinite support of my father Mr. S. Raghu also an advocate in the same subject matter. He was the one who pulled me out of extreme rough weather in my life and stood as a shining lighthouse amidst all storm. His everlasting smile is a great inspiration during all my endeavors. I would like to extend my greatest regards to my mother Mrs. S. Anuradha who has taken all the pain in life to support me during all my endeavors. It would be incomplete if I did not mention the role of my brother Jayanth Raghu who stood like a ladder for me to climb to the top guiding me and supporting me at every step. I still remember the first day of the under graduation where he remarked " , Python , Google " (Brother, learn python well, it will help you get into Google - Original in Kannada). I would also like to thank my professors Dr. A. Srinivas, Prof. NS Kumar, Prof. Ramamurthy Sreenath, Prof. Chidambara Kalamanji, Prof. Nagegowda KS, Prof. D Krupesha, Prof. Rajkumar Rangaswamy, Dr. Juergen Ehrensberger, Dr.Chan Mun Choon, Rajith Noojibial and Prof. Nitin.V.Pujari. I would like to also mention my friends Umesh Rao, Pranav HU, Siddharth Goel, Veeresh Hiremath, Raveendra Pai, Praveen Raj, Praveen Poojari, Prashanth Rajendran and Rajeev Ravi, Younggun Kim, Sony Valdez, Matt Lebrun, Zorex

Salvo who have been my prime pillars of support always. I would like to thank Elvis D'Souza and Umair Z Ahmed who helped me to get into the world of Python. I would like to also thank everyone who helped me on the irc channel with all my doubts. Thanks to my mentors Kiran Hathwar, Hariom Srivastava, Kshitij Saxena, Rahul Bendre, Mr. Krishnakumar, Mr. NS Nagaraja, Mr. Madhu Iyengar, Mr. Srinivas Thonse, Mr. Kurt Griffiths, Mr. Alejandro Cabrera, Mr. Fabio Percoco, Supreeth Kumar, Arun Das, Siva Ekambaram. Thanks to my non-technical mentors Vid. S. Venkatesh and Vid. H.S. Venugopal who have been the guiding light in my life. I would like to also acknowledge the support of my cousins Tejaswini Pillai and Sunil Mudambi.

"Sarve Bhavanthu Sukhinaha, Sarve Santhu Niramayaha, Sarve Bhadrani Pashyantu, Maa Kashchith Dukha Bhagbhaveth. Om Shanthi, Shanthi, Shanthihi". "Om, May All become Happy, May All be Healthy (Free from Illness) May All See what is Auspicious, May no one Suffer in any way."

Also incidentally I happened to start writing this book on International Day of Yoga and I would like to extend my humblest regards to all the great saints and philosophers of this country from whom great thoughts have emerged and spread to all over the world today. If anyone asks me why I am proud of my country I would always say " India is the first country which spoke about looking inwards for happiness".

I would like to extend my gratitude to Guido Van Rossum without whose hard work and sacrifice such a wonderful programming language would not have existed.

My gratitude also goes to the team at BPB Publications for being supportive enough and providing me enough time to complete the book. Lastly, thanks to Rajat Singh for the in-depth technical review of the book whose comments have helped shape the readability of the book.

Preface

Going through the source code of a project involves exhaustive efforts and no planned steps to going through the knowledge base. Understanding the source code provides clear and descriptive knowledge of the workings of the project helping better decision making in our day to day work. When I started the journey back in 2014 to understand the source code of python, there were very few materials available to help navigate the same prompting an idea to create an unofficial guide.

The book begins with the basics of the language which are the base objects and expands to understand the objects built on top of the these base objects which are the base types (integer, float, string, etc.) to the iterable objects which are the lists, tuples and dictionaries.

The interpreter forms the core of the python implementation. Through opcodes, the implementation of each of them has been explained in detail.

The implementation of the GIL always intrigues developers on its requirement and impact on day to day applications and if possible any route to removing the same. Although the book covers the requirement and impact of the GIL, the answer to the removal remains unsolved and is left to the discovery of the curious readers.

Chapter 1: The PyObject and PyVarObjects form the core of the python data structures and contain the pointers to the typeobject and the reference count of the variable. The reference count is internally used to deallocate the variable which has also been covered in this chapter. The PyVarObject is the base object for iterable types such as lists, tuples, sets and dictionaries and contain the length of the object along with the PyObject.

Chapter 2: This chapter covers the implementation of basic python types which are the long, float and boolean objects. The Long object contains the implementation as an array of digits and the operations are implemented as normal high school operations which are quite interesting to understand. The float object internally contains a C float data type, which is used to perform the numerical operations.

Chapter 3: The chapter covers the structure and implementation of the iterable types which are the lists and tuples. Lists are implemented as an array of PyObjects while tuples, though similar in operation do not allow manipulation post the creation and hence is a hashable python type. The implementations of common list and tuple operations such as length, index, subscripting are explained in this chapter.

Chapter 4: The chapter covers the structure and implementation of the hashable types which are the sets and dictionaries. Sets and dictionaries are structurally and functionally similar to each other and share similar implementations which can intrigue us as developers. Sets perform O(1) lookup which can be a gamechanger for programs frequently searching for elements within a set of elements.

Chapter 5: The chapter covers the structure and implementation of the functions and generators which structurally contain the code object which is then used to create a stack frame to execute the function call. Similarly, the generator object contains a reference to the last executed instruction, which is used to return back to the point of execution.

Chapter 6: The chapter covers the implementation of memory management in python which includes the creation and allocation of memory to objects using pre-allocated memory chunks referred to as arenas. The chapter further delves into the details on how memory is allocated by further dividing the arenas into pools and deriving the memories from these chunks. The chapter also covers on how the objects are reassigned back into the pools.

Chapter 7: Interpreter and Opcodes form the fundamentals of the executable parts of a python program. A python program is converted into an executable opcode which is then executed within an interpreter loop which executes each of these opcodes within the giant loop. The interpreter also contains hacks to speed up the execution of these opcodes.

Chapter 8: The chapter can intrigue a lot of developers to understand the functioning of the GIL and its impact on the programming language and how the GIL can be used for achieving better performant systems. The chapter covers the GIL by understanding the process of thread creation and how it impacts the performance.

Chapter 9: Async functions form the core of the async programming which aims at creating single threaded applications without external framework support considering the adoption of async frameworks and performance benefits provided. The chapter explains how each of these are a natural extension of the standard generator object.

Chapter 10: This chapter contains the basics of the source code structure and the implementations of the parser, syntax tree generator and assembler. The chapter serves to introduce these concepts as delving into them will be a completely new book.

Downloading the coloured images:

Please follow the link to download the
Coloured Images of the book:

https://rebrand.ly/k7je3ln

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at business@bpbonline.com for more details.

At you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

BPB IS SEARCHING FOR AUTHORS LIKE YOU

If you're interested in becoming an author for BPB, please visit www.bpbonline.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

The code bundle for the book is also hosted on GitHub at In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at Check them out!

PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

IF YOU ARE INTERESTED IN BECOMING AN AUTHOR

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit

Table of Contents

CHAPTER 1

Design of Generic Objects

The components of a generic Python object contain the memory layout, operations, and memory management. This chapter covers the similarities and differences between Python types. The chapter begins with the **PyObject** and explains how it encapsulates the type, reference count, and bookkeeping pointers. The **PyVarObject,** which is a **PyObject** encapsulation with the size of the data structure, is covered along with the different functional attributes of the type such as numerical, sub stringing, and so on.

Structure

In this chapter, we will cover the following topics:

The **PyObject**

Understanding **_PyObject_HEAD_EXTRA**

Reference counting

The **PyVarObject**

The **PyTypeObject**

Generic type function prototypes

Specific type function prototypes

The Type object substructures

The **PyNumberMethods** substructure

The **PySequenceMethods** substructure

The **PyMappingMethods** substructure

The type object

Name and sizes of types

Allocator, deallocator, and initialization functions

Iterator functions

Methods and attributes

Objective

After studying this chapter, you will be able to understand the basic components, the object and the type, which compromise both the data types and data structures. You will also learn about the **TypeObject** that contains the operations implemented in the data types.

The PyObject

The **PyObject** contains the details of all Python objects and understanding its structure is crucial before we delve into the implementation of reference counting and Python types, which are key elements of this generic object:

Include/object.h Line no 104

```
typedef struct _object {
_PyObject_HEAD_EXTRA
Py_ssize_t ob_refcnt;
struct _typeobject *ob_type;
} PyObject;
```

The **PyObject** contains three elements, which are as follows:

A macro that expands to track all objects in the Python heap in debug mode.

An integer value that holds the reference count of the object.

A pointer to the type of the object such as integer/list/dictionary, which contains the operations that can be performed on an instance of the type.

The upcoming chapters contain both the implementation and structure of each of these elements.

Understanding __PyObject_HEAD_EXTRA

The **_PyObject_HEAD_EXTRA** adds a forward and reverse pointer to every object used to construct a doubly linked list that tracks live objects in the Python heap at runtime. This flag has to be enabled when building the Python executable using the following command:

../configure CFLAGS='-DPy_DEBUG -DPy_TRACE_REFS' --with-pydebug

The code block explaining the definition of the **_PyObject_HEAD_EXTRA** macro is as follows:

Include/object.h Line no 67

```
#ifdef Py_TRACE_REFS -> 1
/* Define pointers to support a doubly-linked list of all live heap objects. */
#define _PyObject_HEAD_EXTRA                    \
struct _object *_ob_next;   -> 2                \
struct _object *_ob_prev;   -> 3

#define _PyObject_EXTRA_INIT 0, 0,

#else
```

```
#define _PyObject_HEAD_EXTRA
#define _PyObject_EXTRA_INIT
#endif
```

Code insights are as

The **_PyObject_HEAD_EXTRA** adds the pointers only when Python is compiled with the **Py_TRACE_REFS** flag enabled.

The **_ob_next** pointer points to the next created object in the linked list.

The **_ob_prev** pointer points to the previously created object in the linked list.

The following code block explains the construction of the live object heap as a doubly-linked list:

Objects/object.c (Line no 81)

```
static PyObject refchain = {&refchain, &refchain};
void _Py_AddToAllObjects(PyObject *op, int force) {
....
if (force || op->_ob_prev == NULL) {
op->_ob_next = refchain._ob_next;
op->_ob_prev = &refchain;
refchain._ob_next->_ob_prev = op;
refchain._ob_next = op;
}
```

.....
}

The highlighted part explains the construction of the doubly-linked list with reference to the current pointer to the end of the list that is pointed to by

This reference chain can be accessed using the **getobjects** method of the **sys** module. The sample Python code demonstrates how the objects can be fetched using the method:

```
>>> import sys
>>> objs = sys.getobjects(1) # 1 returns the first object from the linked list.
>>> more_objs = sys.getobjects(20) # Increase the count to return more objects.
>>> type_objs = sys.getobjects(20, str) # A second argument
```

The following code block explains the implementation of the getobjects function.

Objects/object.c line no 1929

```
PyObject * _Py_GetObjects(PyObject *self, PyObject *args)
{
int i, n;
PyObject *t = NULL;
PyObject *res, *op;
if (!PyArg_ParseTuple(args, "i|O", &n, &t))
```

```
return NULL;
op = refchain._ob_next;
res = PyList_New(0);
if (res == NULL)
return NULL;
for (i = 0; (n == 0 || i < n) && op != &refchain; i++) {
while (op == self || op == args || op == res || op == t ||
(t != NULL && Py_TYPE(op) != (PyTypeObject *) t)) {
op = op->_ob_next;
if (op == &refchain)
return res;
}
if (PyList_Append(res, op) < 0) {
Py_DECREF(res);
return NULL;
}
op = op->_ob_next;
}
return res;
}
```

Reference counting

Python uses reference counting to track the usage of an object and for removing it from the heap on completion of usage. Every object in Python contains the **ob_refcnt** variable, which contains the current reference count. The reference count is incremented every time the object is used such as adding to a list and decrementing when the usage is completed. Once the reference count reaches the object is removed from the heap using the custom deallocator of the type.

The following Python program demonstrates how the **getrefcount** method in the **sys** module can return the current reference count of the object:

```
>>> import sys
>>> a = 1
>>> sys.getrefcount(a)
114
```

The following Python program demonstrates how the **getrefcount** method in the **sys** module can track changes to the reference count:

```
>>> import sys
>>> a = 20000000
>>> sys.getrefcount(a)
```

```
2
>>> b = [a, a]
>>> sys.getrefcount(a)
4
>>> b.pop()
>>> sys.getrefcount(a)
3


>>> b.pop()
>>> sys.getrefcount(a)
2
>>> def f(a):
print(sys.getrefcount(a))


>>> a = 2000000
>>> sys.getrefcount(a)
2
>>> f(a)
4
```

The function returns **4** because the **sys.refcount** function also increments the reference count by

The following C Python code handles reference counting:

```
typedef struct _object {
_PyObject_HEAD_EXTRA
Py_ssize_t ob_refcnt;
struct _typeobject *ob_type;
} PyObject;
```

The **ob_refcnt** variable in the **PyObject** structure holds the reference count of every **PyObject,** which is manipulated using two macros **_Py_INCREF** and

The code block explaining incrementing of reference counting is as follows:

```
static inline void _Py_INCREF(PyObject *op)
{
_Py_INC_REFTOTAL;
op->ob_refcnt++;
}

#define Py_INCREF(op) _Py_INCREF(_PyObject_CAST(op))
```

The macro **Py_INCREF** increments the reference count of the variable by

```
static inline void _Py_DECREF(const char *filename, int lineno,
PyObject *op)
{
(void)filename; /* may be unused, shut up -Wunused-parameter */
(void)lineno; /* may be unused, shut up -Wunused-parameter */
_Py_DEC_REFTOTAL;
if (--op->ob_refcnt != 0) {
#ifdef Py_REF_DEBUG
if (op->ob_refcnt < 0) {
_Py_NegativeRefcount(filename, lineno, op);
```

```
}
#endif
}
else {
_Py_Dealloc(op);
}
}
```

```
#define Py_DECREF(op) _Py_DECREF(__FILE__, __LINE__,
_PyObject_CAST(op))
```

Code insights are as

Decrement the reference count when the requestor completes usage of the object.

Deallocate the object when the reference count becomes The deallocator depends on the type of the object, which will be covered in the subsequent chapters on basic and iterable types.

The PyVarObject

The **PyVarObject** is the generic container object that holds the data for lists, types, dictionaries, and sets:

Objects/object.h Line no 113

```
typedef struct {
PyObject ob_base;
Py_ssize_t ob_size; /* Number of items in variable part */
} PyVarObject;
```

The **PyVarObject** contains a **PyObject** for data storage and the size of the container.

The macro **Py_SIZE** is used to fetch the current size of the container when working with them.

The following code block explains the functioning of

```
#define Py_SIZE(ob) (_PyVarObject_CAST(ob)->ob_size)
```
Objects/object.h Line no 96

```
#define PyObject_VAR_HEAD PyVarObject ob_base;
```
Include/tupleobject.h Line no 9

```
typedef struct {
PyObject_VAR_HEAD
/* ob_item contains space for 'ob_size' elements.
Items must normally not be NULL, except during construction when
the tuple is not yet visible outside the function that builds it. */
PyObject *ob_item[1];
} PyTupleObject;
```

The **PyVarObject** is a part of the **PyTupleObject** and stores the and We will study more about the type structure and data storage of the **PyTupleObject** in the upcoming chapters.

The PyTypeObject

The type of an object encompasses the operations that can be performed on an instance of its type. The possible operations on integers can be and **division** but fetching a substring is invalid, but a valid operation on a string or a list. The **PyTypeObject** contains the references to valid operations on the type and marks the others as irrelevant.

The subchapters cover the skeletal structure of the type object before understanding the operations in them.

Generic type function prototypes

Type object function prototypes are classified depending on the number of arguments as and **ternaryfunc** based on the argument count and on the basis of the operation of the function, such as and so on.

Include/object.h (Line no 140)
typedef PyObject * (*unaryfunc)(PyObject *);

The **unaryfunc** prototype accepts one parameter and returns a pointer to a

An example is as follows:

unaryfunc nb_negative;

Changing a number to negative accepts a **PyObject** (integers/float) and returns the negative number for the same.

The **binaryfunc** prototype accepts two **PyObject** parameters and returns a pointer to a

typedef PyObject * (*binaryfunc)(PyObject *, PyObject *,);

An example is as follows:

binaryfunc nb_add;

The **nb_add** function is defined for all types that support the addition operation such as integers, floating point numbers, strings, and container types such as lists, dictionaries, sets, and so on.

The **ternaryfunc** is a prototype for all type functions, which accept three parameters for operations and returns a pointer to a **PyObject** as the return value:

typedef PyObject * (*ternaryfunc)(PyObject *, PyObject *, PyObject *);

The example is as follows:

ternaryfunc nb_power;
>>> pow(4, 4)
256

The **pow()** built in function raises an element to the power of another number defined as the second parameter. The function also accepts an optional third parameter, which is the modulo of the result.

The example is as follows:

```
>>> pow(4, 4, 5)
1
```

The code corresponds to the mathematical equation => *4^4 % 5 = 256 % 5 ==*

```
typedef int (*inquiry)(PyObject *);
```

The **inquiry** is the prototype for all functions that takes a **PyObject** and returns an integer value in response. The **lenfunc** returns the length of all iterable types:

```
typedef Py_ssize_t (*lenfunc)(PyObject *);
```

The example is as follows:

Objects/listobject.c Line no 2797

```
(lenfunc)list_length, /* sq_length */
```

```
>>> l = [1, 2, 3]
>>> len(l)
3
```

The **list_length** computes the length of the input list and returns the size as the return value.

The **ssizeargfunc** prototype accepts a **Py_ssize_t** argument along with the **PyObject** and returns a **PyObject** in response:

typedef PyObject *(*ssizeargfunc)(PyObject *, Py_ssize_t);

The example is as follows:

Objects/listobject.c Line no 2798
(ssizeargfunc)list_repeat, /* sq_repeat */
>>> l = [1, 2, 3]
>>> **l2 = l * 3 # List repeat function**
>>> l2
[1, 2, 3, 1, 2, 3, 1, 2, 3]

The **list_repeat** function takes a Python list and creates a new list that has *n* repetitions of the original list.

The **ssizeobjargprocfunc** function prototype accepts a **Py_ssize_t** and two **PyObjects** and returns an integer:

typedef int(*ssizeobjargproc)(PyObject *, Py_ssize_t, PyObject *);

The example is as follows:
Objects/listobject.c (2801)
(ssizeobjargproc)list_ass_item, /* sq_ass_item */

The function **list_ass_item** adds an element at the particular index in a Python list.

The **objobjargproc** prototype takes three **PyObject** arguments and returns an integer value in response:

typedef int(*objobjargproc)(PyObject *, PyObject *, PyObject *);
Object/dictobject.c Line no 2135

The example is as follows:

(objobjargproc)dict_ass_sub, /*mp_ass_subscript*/

The function **dict_ass_sub** inserts an *element (1)* into the *dictionary (2)* at the *index*

The **objobjproc** prototype takes two **PyObject** parameters and returns an integer:

typedef int (*objobjproc)(PyObject *, PyObject *);

The example is as follows:

Objects/listobject.c line no 2803
(objobjproc)list_contains, /* sq_contains */

The **list_contains** function checks whether the second object exists in the list and returns the index where it is present or **-1** if the element is not present. We will cover more about this in the *chapter 3, on Iterable Sequence*

Specific type function prototypes

The previous section covered the generic prototypes, which constitute type functions having specific input and output parameter signatures. This section discusses the prototypes for specific purposes such as freeing an object/traversing an iterable type:

Include/object.h Line no 152
typedef int (*visitproc)(PyObject *, void *);
typedef int (*traverseproc)(PyObject *, visitproc, void *);

The **traverseproc** prototype is used in all container types to iterate over the elements in the object.

The example is as follows:

Objects/listobject.c line no 3053
(traverseproc)list_traverse, /* tp_traverse */
static int list_traverse(PyListObject *o, visitproc visit, void *arg)
{
Py_ssize_t i;
for (i = Py_SIZE(o); --i >= 0;)
Py_VISIT(o->ob_item[i]);
return 0;
}

The **list_traverse** method traverses every item in the list and returns the particular element to perform the operation passed at **visitproc** to this method.

The **destructor** function prototype frees the memory allocated to the object after being used:

typedef void (*destructor)(PyObject *);

The following code block demonstrates how the destructor is called when the reference count reaches

Include/cpython/object.h Line no 339
static inline void _Py_Dealloc_inline(PyObject *op)
{
destructor dealloc = Py_TYPE(op)->tp_dealloc;
#ifdef Py_TRACE_REFS
_Py_ForgetReference(op);
#else
_Py_INC_TPFREES(op);
#endif
(*dealloc)(op);
}

#define _Py_Dealloc(op) _Py_Dealloc_inline(op)

The **getattrfunc/setattrfunc** prototype is used to fetch/set an attribute of the type using the built-in functions:
typedef PyObject *(*getattrfunc)(PyObject *, char *);

```
typedef PyObject *(*getattrofunc)(PyObject *, PyObject *);
typedef int (*setattrfunc)(PyObject *, char *, PyObject *);
typedef int (*setattrofunc)(PyObject *, PyObject *, PyObject *);
```

The functions being generic in nature are assigned during type initialization to a generic getter and setter function.

The **reprfunc** converts the object to a human-readable string representation:

```
typedef PyObject *(*reprfunc)(PyObject *);
```

The example is as follows:

```
static PyObject *list_repr(PyListObject *v)
{

...
if (_PyUnicodeWriter_WriteChar(&writer, '[') < 0)
goto error;
/* Do repr() on each element. Note that this may mutate the list,
so must refetch the list size on each iteration. */
for (i = 0; i < Py_SIZE(v); ++i) {
if (i > 0) {
if (_PyUnicodeWriter_WriteASCIIString(&writer, ", ", 2) < 0)
goto error;
}

s = PyObject_Repr(v->ob_item[i]);
if (s == NULL)
```

```
goto error;

if (_PyUnicodeWriter_WriteStr(&writer, s) < 0) {
Py_DECREF(s);
goto error;
}
Py_DECREF(s);
}

writer.overallocate = 0;
if (_PyUnicodeWriter_WriteChar(&writer, ']') < 0)
goto error;
...
}
>>> l = [1, 2, 3]
>>> l
[1, 2, 3]
```

The highlighted lines in the **repr** function for a list print followed by the representation of every object in the list and finally, character to the **stdout** as seen in the preceding Python example:

```
typedef Py_hash_t (*hashfunc)(PyObject *);
```

The **hashfunc** converts the object to a unique hash. This unique hash can be used as a key for a dictionary among several uses to convert an object into a unique integer representation. Unmutable types like strings, integers, tuples implement this function, whereas mutable types like lists, dictionaries do not implement it.

Code block demonstration the definitions of the hashing function in the list and tuple types:

Objects/tupleobject.c (Line no 843)
(hashfunc)tuplehash, /* tp_hash */

Objects/listobject.c (Line no 3044)
PyObject_HashNotImplemented, /* tp_hash */

The **richcmpfunc** prototype compares two objects of the same type and checks for equality:

typedef PyObject *(*richcmpfunc) (PyObject *, PyObject *, int);

The example is as follows:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a == b
True
```

Objects/listobject.c Line no 3055
list_richcompare, /* tp_richcompare */

The **list_richcompare** method compares two lists for equality by checking if every element is equal. The chapter on lists explains the implementation of this function in detail.

Code block demonstrates the definition of the **getiterfunc** function in the list and tuple types:

```
typedef PyObject *(*getiterfunc) (PyObject *);
```

Objects/listobject.c (Line no 3057)

```
list_iter, /* tp_iter */
```

Objects/listobject.c (Line no 3134)

```
static PyObject* list_iter(PyObject *seq)
{
listiterobject *it;

if (!PyList_Check(seq)) {
PyErr_BadInternalCall();
return NULL;
}
it = PyObject_GC_New(listiterobject, &PyListIter_Type);
if (it == NULL)
return NULL;
it->it_index = 0;
Py_INCREF(seq);
it->it_seq = (PyListObject *)seq;
_PyObject_GC_TRACK(it);
return (PyObject *)it;
}
```

The **list_iter** fetches the iterator for the list object. In the preceding code snippet, we see that it returns a **listiterobject** that is initialized to **0** corresponding to the first index of the array.

The following code block demonstrates the definition and implementation of the **iternextfunc** function in the list and tuple types:

```
typedef PyObject *(*iternextfunc) (PyObject *);
Objects/listobject.c (Line no 3134)
(iternextfunc)listiter_next, /* tp_iternext */
static PyObject* listiter_next(listiterobject *it)
{
PyListObject *seq;
PyObject *item;

assert(it != NULL);
seq = it->it_seq;
if (seq == NULL)
return NULL; // -> 3
assert(PyList_Check(seq));

if (it->it_index < PyList_GET_SIZE(seq)) {
item = PyList_GET_ITEM(seq, it->it_index);
++it->it_index;
Py_INCREF(item);
return item;
} // -> 1
```

```
it->it_seq = NULL;
Py_DECREF(seq); // -> 2
return NULL;
}
```

The preceding code block explains list iteration implemented by the

Iterate the list until the index in the iterator is equal to the size of the list.

Assign the list pointed by the iterator to **NULL** when the index **it_index == size of the**

Terminate the iteration when the list pointed is The interpreter will raise the **StopIterationException.**

The type object substructures

Sections on PyNumberMethods and *PySequenceMethods* cover generic function prototypes, which constitute the type of a Python object. This section covers the substructures of the **type** object that provide particular function sets such as numerical/iteration to the type.

The PyNumberMethods substructure

The **PyNumberMethods** structure encapsulates all numerical operations on a type and is primarily implemented by numerical types. This structure also encapsulates methods to handle inline operations such as and so on:

Objects/cpython/object.h (Line no 95)

```
typedef struct {
/* Number implementations must check *both*
arguments for proper type and implement the necessary conversions
in the slot functions themselves. */

binaryfunc nb_add;
binaryfunc nb_subtract;
binaryfunc nb_multiply;
binaryfunc nb_remainder;
binaryfunc nb_divmod;
ternaryfunc nb_power;
unaryfunc nb_negative;
unaryfunc nb_positive;
unaryfunc nb_absolute;
inquiry nb_bool;
unaryfunc nb_invert;
binaryfunc nb_lshift;
```

```
binaryfunc  nb_rshift;
binaryfunc  nb_and;
binaryfunc  nb_xor;
binaryfunc  nb_or;
...
}  PyNumberMethods;
```

The following code block demonstrates the numerical operations on the **long** object defined in the **PyNumberMethods** structure:

Objects/longobject.c (Line no 5678)

```
static PyNumberMethods long_as_number = {
(binaryfunc)long_add,          /*nb_add*/
(binaryfunc)long_sub,          /*nb_subtract*/
(binaryfunc)long_mul,          /*nb_multiply*/
long_mod,                        /*nb_remainder*/
long_divmod,                     /*nb_divmod*/
long_pow,                        /*nb_power*/
(unaryfunc)long_neg,           /*nb_negative*/
long_long,                       /*tp_positive*/
...
};
```

The **LongType** implements most of the numerical operations while the list type marks the same as indicating that numerical operations are not valid on lists:

Objects/listobject.c (Line no 3030)

```
o,  /* tp_as_number */
```

The PySequenceMethods substructure

```
typedef struct {
lenfunc sq_length;
binaryfunc sq_concat;
ssizeargfunc sq_repeat;
ssizeargfunc sq_item;
void *was_sq_slice;
ssizeobjargproc sq_ass_item;
void *was_sq_ass_slice;
objobjproc sq_contains;

binaryfunc sq_inplace_concat;
ssizeargfunc sq_inplace_repeat;
} PySequenceMethods;
```

The **PySequenceMethods** defines functions related to iterable types such as length, concatenation, repetition, slicing, and so on.

The following code block demonstrates the definition of the **SequenceMethods** in the list and tuple types:

```
Objects/listobject.c (Line no 2795)
static PySequenceMethods list_as_sequence = {
(lenfunc)list_length,                    /* sq_length */
(binaryfunc)list_concat,                 /* sq_concat */
(ssizeargfunc)list_repeat,               /* sq_repeat */
```

```
(ssizeargfunc)list_item,                /* sq_item */
0,                                      /* sq_slice */
(ssizeobjargproc)list_ass_item,     /* sq_ass_item */
0,                                      /* sq_ass_slice */
(objobjproc)list_contains,              /* sq_contains */
(binaryfunc)list_inplace_concat,    /* sq_inplace_concat */

(ssizeargfunc)list_inplace_repeat,  /* sq_inplace_repeat */
};
```

Code block demonstrates sequence methods attribute being assigned to **NULL** for the **long** type:

Objects/longobject.c (line no 5727)

```
0,  /* tp_as_sequence */
```

The PyMappingMethods substructure

```
typedef struct {
lenfunc mp_length;
binaryfunc mp_subscript;
objobjargproc mp_ass_subscript;
} PyMappingMethods;
```

The **PyMappingMethods** structure defines operations for length, subscript, and the operations, which act on the list for mainly the operators related to the index operator

```
static PyMappingMethods list_as_mapping = {
(lenfunc)list_length,
(binaryfunc)list_subscript,
(objobjargproc)list_ass_subscript
};
```

Although the **PyMappingMethods** are defined for a list it remains undefined for a **long** object:

Objects/longobject.c (Line no 5728)

```
0, /* tp_as_mapping */
```

Code block demonstrates sequence methods attribute being assigned to **NULL** for the **long** type.

The type object

The previous sections covered the subtypes that compose type object. This section covers the different parts of the type object. This chapter will be split up into many sections for simplicity.

Name and sizes of type

Objects/cpython/object.h (Line no 177)

```
typedef struct _typeobject {
PyObject_VAR_HEAD
const char *tp_name; /* For printing, in format "." */
Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */
...
}
```

The **tp_name** is a human-readable version of the type name as a string.

Code block demonstrating the name of the long object as

```
PyTypeObject PyLong_Type = {
PyVarObject_HEAD_INIT(&PyType_Type, 0)
"int", /* tp_name */
```

```
>>> a = 20
>>> type(a)
'int'>
```

The **type** object contains two sizes, the **tp_basicsize** and the **tp_itemsize,** which indicate the fixed and variable sizes of the

variable type. The sizes are used to request the appropriate memory from the memory allocator.

Code block demonstrating the allocation of memory to an object using the basic and item sizes:

Include/objectimpl.h (Line no 179)

```
#define _PyObject_VAR_SIZE(typeobj, nitems)        \
_Py_SIZE_ROUND_UP((typeobj)->tp_basicsize + \
(nitems)*(typeobj)->tp_itemsize,              \
SIZEOF_VOID_P)
```

We see that the requested memory is the *base size of the object + (n * the size of the item) of every*

Allocator, deallocator, and initialization functions

destructor tp_dealloc;
initproc tp_init;
allocfunc tp_alloc;

The deallocator function **tp_dealloc** is called when the reference count of the variable becomes

Most standard Python types do not implement these functions and use the generic memory allocation methods defined for all Python objects.

Iterator functions

The **tp_iter** function creates and returns an iterable object for a type, whereas the **tp_iternext** function handles the navigation of the elements in the iterator. For most iterable types, the **tp_iter** method is implemented in the base type, whereas the **tp_iternext** method is implemented by an iterable type for the class. For example, the **PyListType** implements the **tp_iter** method and returns an object of the type **PyListIter_Type,** which handles the navigation of elements:

```
/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;
```

Code block illustrating the construction of an iterator for the **tuple** object:

Objects/tupleobject.c (Line no 856)

```
tuple_iter, /* tp_iter */

static PyObject * tuple_iter(PyObject *seq)
{
tupleiterobject *it;
```

```
if (!PyTuple_Check(seq)) {
PyErr_BadInternalCall();
return NULL;
}
it = PyObject_GC_New(tupleiterobject, &PyTupleIter_Type);
if (it == NULL)
return NULL;
it->it_index = 0;
Py_INCREF(seq);
it->it_seq = (PyTupleObject *)seq;

_PyObject_GC_TRACK(it);
return (PyObject *)it;
}
```

The **tuple_iter** returns an object of the type **tupleiterobject** that implements the **tupleiter_next** (implementation of function that handles the iteration on the tuple.

Methods and attributes

A type encapsulates the data and the operations that can be performed on it. In this section, we will cover the structure of both data and operations that can be added to a Python type. The type object contains the **PyMethodDef** array that stores the possible operations on the type, and the **PyMemberDef** array that encapsulates the data for the object:

```
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
```

Code block illustrating the attributes of the **PyMethodDef** structure:

```
struct PyMethodDef {
const char  *ml_name;    /* The name of the built-in function/method */
PyCFunction ml_meth;     /* The C function that implements it */
int         ml_flags;    /* Combination of METH_xxx flags, which mostly describe the args expected by the C func */
const char  *ml_doc;     /* The __doc__ attribute, or NULL */
};
```

The definition includes the name of the method, along with the implementation stored in **ml_meth,** and the documentation that is printed when requested:

```
typedef struct PyMemberDef {
const char *name;
int type;
Py_ssize_t offset;
int flags;

const char *doc;
} PyMemberDef;
```

Code insights are as

const char a string that holds the name of the member to be accessed.

int an integer denoting the type of the member. The valid types are defined in **Include/structmember.h** *no*

Py_ssize_t an integer denoting the memory offset to fetch the content of the member. The offset is calculated using the **offsetof** function from the C standard library.

int the flags indicate the type of operations that can be performed on this member, such as **READONLY** and flags that describe how it must be modified in the restricted mode.

const char string that contains the documentation string for the member.

The members are not part of the type but have individual values in every instance of the type. For most types, the attributes of the

object are exposed as members to be accessed through code. Let us take an example of the complex object and examine the members defined for the type. To understand the members of the type, we will have to examine the attributes of the complex object:

```
typedef struct {
double real;
double imag;
} Py_complex;
```

The **complex** object contains two attributes, the **real** and the **imaginary** part. Let us understand this through a Python sample:

```
>>> l = 8 + 4j
>>> type(l)
'complex'>
>>> l
(8+4j)
>>> l.real
8.0
>>> l.imag
4.0
```

The **complex** object contains two members, **real** and **imag, which** are floating-point representations of the real and imaginary parts of a complex number in Python. Let us now check their definitions in code:

Objects/complexobject.c (Line no 746)

```
static PyMemberDef complex_members[] = {
{"real", T_DOUBLE, offsetof(PyComplexObject, cval.real),
READONLY,
"the real part of a complex number"},
{"imag", T_DOUBLE, offsetof(PyComplexObject, cval.imag),
READONLY,
"the imaginary part of a complex number"},
{0},
};
```

Printing an instance of the type

As instances of a type are areas in memory, it is necessary to represent it in a human-readable format. Every type contains a function that indicates how it can be converted to a human-readable string format, which is useful to print on the console/in logs.

reprfunc tp_str;

An example is as follows:

In this example, we will cover the representation function of the **complex** object using a simple Python example:

```
>>> a = 4 + 9j
>>> a
(4+9j)
```

When we printed the **complex** object, the format of the printed string was *realpart* + *imagpart* The following code shows how the object is printed in this format:

Objects/complexobject.c (line no 353)

```
static PyObject* complex_repr(PyComplexObject *v)
```

```
{
....
} else {
/* Format imaginary part with sign, real part without. Include
parens in the result. */
pre = PyOS_double_to_string(v->cval.real, format_code, precision, 0,
NULL);
if (!pre) {
PyErr_NoMemory();
goto done;
}

re = pre;

im = PyOS_double_to_string(v->cval.imag, format_code, precision,
Py_DTSF_SIGN, NULL);
if (!im) {
PyErr_NoMemory();
goto done;
}
lead = "(";
tail = ")";
}
result = PyUnicode_FromFormat("%s%s%sj%s", lead, re, im, tail);
....
}
```

Conclusion

This chapter covers the **PyObject** that contains the reference count and the type of the object, along with the pointers to the doubly linked list of the live heap objects for inspection in debug mode.

The **PyVarObject** is the base object for container types that comprises the **PyObject** along with the size of the container.

The reference count of the object is incremented using the **Py_IncRef** macro and decremented using the **Py_DecRef** macro, and on reaching **0** the object is deallocated from the heap.

Python types consist of prototype functions such as **unaryfunc** based on the argument input count and types, along with covering the prototypes for specific operations such as the destructor prototype.

We finally covered the components that form a Python type and how they are linked to form the operations and data stored in the type instance.

The next chapter will cover the structure and operations of the integer, string, Boolean, and floating-point types. We will begin with the memory layout for the data types and cover in brief the implementation of a few operations.

CHAPTER 2

Basic Python Types

In the previous chapter, we covered the structure of a Python type, which contains the display name, size of memory for allocation, and the operations that can be performed on the data type. Type objects also hold methods for memory allocation and freeing of the object to be constructed on user request and destructed when the object is no longer used.

Every user program stores data in variables and the most common types are integers, Boolean, strings, and floating-point numbers. This chapter covers the structure, memory management, and operations that can be performed on each type.

Structure

In this chapter, we will cover the following topics:

The Bool object

The Bool type

Creating a Boolean object

Representation of Boolean objects

Operations on Boolean objects

The Long object

The type of the long object

Creating a new long object

Arithmetic operations

Bitwise operations

The Float object

The type object

Creating a new float object

Arithmetic operations

The None object

Objectives

After completing this unit, you will be able to understand the structure and operations of basic Python types, that is, and the **None** object type.

The Bool object

Boolean objects contain the result of logical operation to be either **True** or Programming languages capture this construct in many different ways, although in compiled languages such as C/C++, any value greater than *0* is considered whereas the value *0* is considered Languages such as Python store the Boolean value as integers. The specific storage scheme for Boolean values in Python is shown as follows.

The following code block demonstrates the structure of the Boolean object:

Objects/boolobject.c (Line no 177)

```
struct _longobject _Py_FalseStruct = {
PyVarObject_HEAD_INIT(&PyBool_Type, 0)
{0} // -> 1
};

struct _longobject _Py_TrueStruct = {
PyVarObject_HEAD_INIT(&PyBool_Type, 1)
{1} // -> 2
};
```

Include/boolobject.h (Line no 21)

#define Py_False ((PyObject *) &_Py_FalseStruct) // -> 3
#define Py_True ((PyObject *) &_Py_TrueStruct) // -> 4

Python/bltinmodule.h (Line no 2809)

SETBUILTIN("False", Py_False); // -> 5
SETBUILTIN("True", Py_True); // -> 6

Code insights are as

The value **False** is stored as a long object with value

The value **True** is stored as a long object with value

The macro **Py_False** is created as the address of the **_Py_FalseStruct.**

The macro **Py_True** is created as the address of the

The macro **Py_False** is saved as the built-in **False** for use in programs without importing.

The value **Py_True** is saved as the built-in **True** for use in programs without importing.

The Bool type

The type of the Boolean object describes the name, size, and all valid operations that can be performed on it. Boolean objects contain a very small operation set and do not implement any of the numerical/iteration operations like other Python types.

The following code block demonstrates the structure of the Boolean type:

Objects/boolobject.c (Line no 134)

```
PyTypeObject PyBool_Type = {
PyVarObject_HEAD_INIT(&PyType_Type, 0)
"bool", // -> 1
sizeof(struct _longobject), // -> 2
....
bool_repr,                                  /* tp_repr */ // -> 3
&bool_as_number,                            /* tp_as_number */ // -
> 4
....
....
&PyLong_Type,                               /* tp_base */ // -> 5
....
bool_new,                                   /* tp_new */ // -> 6
};
```

Code insights are as

The type of the object is stored as the string This is obtained when we use the type built-in function on a Boolean object.

The size of the Boolean object is defined by the size of the **LongObject,** as we have seen in the previous section that a Boolean object is stored internally as a long value.

The function used for converting a Boolean object to a string representation.

The structure contains all the numerical operations possible on the Boolean type.

The base class for Boolean type is the **PyLong_Type,** as we have seen in the previous section.

The constructor method for the Boolean object.

Creating a Boolean object

Each data structure has its own Boolean representation; for example, any value greater than **0** is considered **True,** whereas **0** is considered **False** for integer types while the length of the list/tuple determines its Boolean value. The **bool** built-in function converts a Python object into its Boolean representation.

The following code block demonstrates the implementation of the construction of a Boolean object:

Objects/boolobject.c (line no 42)

```
static PyObject* bool_new(PyTypeObject *type, PyObject *args,
PyObject *kwds)
{
PyObject *x = Py_False;
long ok;
...
ok = PyObject_IsTrue(x); // -> 1
if (ok < 0)
return NULL;
return PyBool_FromLong(ok); // -> 2
}
```

Objects/object.c (line no 1448)

```
int PyObject_IsTrue(PyObject *v)
{
Py_ssize_t res;
if (v == Py_True) // -> 3
return 1;
if (v == Py_False) // -> 4
return 0;
if (v == Py_None) // -> 5


return 0;
else if (v->ob_type->tp_as_number != NULL &&
v->ob_type->tp_as_number->nb_bool != NULL)
res = (*v->ob_type->tp_as_number->nb_bool)(v); // -> 6
else if (v->ob_type->tp_as_mapping != NULL &&
v->ob_type->tp_as_mapping->mp_length != NULL)
res = (*v->ob_type->tp_as_mapping->mp_length)(v); // -> 7
else if (v->ob_type->tp_as_sequence != NULL &&
v->ob_type->tp_as_sequence->sq_length != NULL)
res = (*v->ob_type->tp_as_sequence->sq_length)(v); // -> 8
else
return 1;
/* if it is negative, it should be either -1 or -2 */
return (res > 0) ? 1 : Py_SAFE_DOWNCAST(res, Py_ssize_t, int); //
-> 9
}
```

Objects/boolobject.c (line no 28)

```
PyObject *PyBool_FromLong(long ok)
{
PyObject *result;
```

```
if (ok)
result = Py_True; // -> 10
else
result = Py_False; // -> 11
Py_INCREF(result);
return result;
}
```

Code insights are as

The function **PyObject_IsTrue** is used to determine if the object equates to Each Python object type has its own way of determining its Boolean nature. For integers values greater than **0** are considered **true,** whereas value **0** is considered to be Empty iterables, such as lists and tuples, are considered false, whereas those with elements are considered true. The behavior is similar for mapped objects such as sets and dictionaries.

The Boolean value of the input object is the return value of the function

If the value is already **Py_True** return **1,** indicating the value equating to

If the value is already **Py_False** return **0,** indicating the Boolean value equating to

If the value is **Py_None** return **0,** indicating the Boolean value equating to

If the value is a numerical type, check for the implementation of the **tp_as_number** methods, and if an implemented check for the implementation of the **nb_bool** function prototype. If implemented, return the value of the type as Boolean.

If the value is a mappable type, check for the implementation of the **tp_as_mapping** methods, and if an implemented check for the implementation of the **mp_length** function prototype. If implemented, return the value of the type as Boolean.

If the value is an iterable type, check for the implementation of the **tp_as_sequence** methods, and if an implemented check for the implementation of the **sq_length** function prototype. If implemented, return the value of the type as Boolean.

Return the result of the function as the integer value **0** or

If the result of the call to the function **PyObject_IsTrue** is return the value

If the result of the **PyObject_IsTrue** is not return

Representation of Boolean objects

Representation functions convert a Python data type into a string that can be used for displaying/printing. For example, programs serialize data to strings to be stored in an external store such as a file or a datastore. Representation functions help convert the data type to a string to be used in such cases.

The following code block demonstrates the implementation of the representation function:

```
Objects/boolobject.c (line no 12)
static PyObject* bool_repr(PyObject *self)
{
PyObject *s;

if (self == Py_True)
s = true_str ? true_str :
(true_str = PyUnicode_InternFromString("True")); // -> 1
else
s = false_str ? false_str :
(false_str = PyUnicode_InternFromString("False")); // -> 2
Py_XINCREF(s);
return s;
}
```

Code insights are as

If the value is equal to **Py_True,** return the string

If the value is equal to **Py_False,** return the string

Operations on Boolean objects

The number of operations that can be performed on Boolean objects being few in number will be covered in a single subsection. Boolean objects support only three kinds of logical operations, which are the logical and

The following code block demonstrates the implementation of the logical AND, OR, and XOR operator:

Objects/boolobject.c (line no 60)

```
static PyObject* bool_and(PyObject *a, PyObject *b)
{
if (!PyBool_Check(a) || !PyBool_Check(b))
return PyLong_Type.tp_as_number->nb_and(a, b); // -> 1
return PyBool_FromLong((a == Py_True) & (b == Py_True)); // -> 2
}
```

Objects/boolobject.c (line no 68)

```
static PyObject* bool_or(PyObject *a, PyObject *b)
{
if (!PyBool_Check(a) || !PyBool_Check(b))
return PyLong_Type.tp_as_number->nb_or(a, b); // -> 3
return PyBool_FromLong((a == Py_True) | (b == Py_True)); // -> 4
}
```

Objects/boolobject.c (line no 76)

```
static PyObject* bool_xor(PyObject *a, PyObject *b)
{
if (!PyBool_Check(a) || !PyBool_Check(b))
return PyLong_Type.tp_as_number->nb_xor(a, b); // -> 5

return PyBool_FromLong((a == Py_True) ^ (b == Py_True)); // -> 6
}
```

Code insights are as

The **nb_and** function prototype on numerical values performs the logical and on the long values. The values, if not Boolean have to be of type **long,** and hence the operation is performed on the **nb_and** method of the **long** object.

The **PyBool_FromLong** returns the Boolean value of the result of **a == Py_True & b == Py_True** when both the values are equal to

The **nb_or** function prototype on numerical values performs the logical or on the long values. The values, if not Boolean have to be of type **long,** and hence the operation is performed on the **nb_or** method of the **long** object.

The **PyBool_FromLong** returns the Boolean value of **a == Py_True | b ==** that is, the result of either of the values being equal to

The **nb_xor** function prototype on numerical values performs the logical XOR on the **long** values.

The **PyBool_FromLong** returns the Boolean value of **a == Py_True ^ b ==**

The Long object

Python supports storing positive and negative numerical values in the **long** object data structure. The data structure supports all numerical operations such as addition, multiplication, subtraction, division, and modulo. It also supports bitwise operators such as AND, OR, and XOR. This section covers the structure, the type, and a few operations to help understand the internal workings. The rest of the operations are left to the curiosity of the readers.

The following code block demonstrates the structure of the **long** object:

Include/longintrepr.h (line no 85)

```
struct _longobject {
PyObject_VAR_HEAD // -> 1
digit ob_digit[1]; // -> 2
};
```

Storage of data in the long object for the number Each number in the **long** object is stored in a digit representation in the form of the array

Code insights are as

The **long** object is considered a type variable as the length of the **long** object is equal to the number of digits it holds.

The data storage is initialized as an array of digits of length

The type of the Long object

The **type** object defines the name, size, and valid operations that can be performed on the data type.

The following code block demonstrates the declaration of the type of the **long** object:

Objects/longobject.c (line no 5715)

```
PyTypeObject PyLong_Type = {
PyVarObject_HEAD_INIT(&PyType_Type, 0) // -> 1
"int",                                  /* tp_name */ // -> 2
offsetof(PyLongObject, ob_digit),       /* tp_basicsize */ // -> 3
sizeof(digit),                          /* tp_itemsize */ // -> 4
...
long_to_decimal_string,                 /* tp_repr   */ // -> 5
...
(hashfunc)long_hash,                    /* tp_hash */ // -> 6
...
long_richcompare,                       /* tp_richcompare */ // -
> 7
...
long_new,                               /* tp_new */ // -> 8
PyObject_Del,                           /* tp_free */ // -> 9
};
```

Code insights are as

Initializing the type of the **long** object to be variable.

The name of the type is represented as the string

The basic size of the **long** object equals the size of all storage, that is, the size, reference count, pointer to the type object, and so on other than the **ob_digit** array, which is the offset to the array.

The size of an item in the list is equal to the size of the digit.

Function to convert a long object to a string representation.

Function to hash the value to be stored as a dictionary key or a set or for other hashing purposes.

Function to compare two long objects for equality.

Constructor function for creating a new **long** object.

Destructor function for deallocation of a **long** object.

Creating a new long object

Allocating memory for a **long** object implies providing storage for all the digits in the object.

The following code block demonstrates the implementation of memory allocation for a **long** object:

Objects/longobject.c (line no 261)

```
PyLongObject* _PyLong_New(Py_ssize_t size)
{
PyLongObject *result;
...
result = PyObject_MALLOC(offsetof(PyLongObject, ob_digit) +
size*sizeof(digit)); // -> 1
...
return (PyLongObject*)PyObject_INIT_VAR(result, &PyLong_Type,
size);
}
```

Code insight is as

The memory required for the **long** object is equal to the *size of the wrapper + the size for all the digits in the long*

Arithmetic operations

Integer objects are primarily used for arithmetic operations such as and so on. These operations are used from the most basic programs to complex data operations. This section covers the **addition** and multiplication operations while the others are left to the curiosity of the readers.

The following code block demonstrates the implementation of the addition operation:

Objects/longobject.c (line no 3151)

```
static PyLongObject* x_add(PyLongObject *a, PyLongObject *b)
{
Py_ssize_t size_a = Py_ABS(Py_SIZE(a)), size_b =
Py_ABS(Py_SIZE(b));
PyLongObject *z;
Py_ssize_t i;
digit carry = 0;
...
z = _PyLong_New(size_a+1); // -> 1

if (z == NULL)
return NULL;

for (i = 0; i < size_b; ++i) {
carry += a->ob_digit[i] + b->ob_digit[i]; // -> 2
```

```
z->ob_digit[i] = carry & PyLong_MASK; // -> 3
carry >>= PyLong_SHIFT; // -> 4
}
for (; i < size_a; ++i) {
carry += a->ob_digit[i]; // -> 5
z->ob_digit[i] = carry & PyLong_MASK; // -> 6


carry >>= PyLong_SHIFT; // -> 7
}
z->ob_digit[i] = carry; // -> 8
return long_normalize(z);
}
```

Code insights are as

Initializing a new **long** object to store the result of the operation.

Compute the sum of two digits at position *i* to the variable carry.

Save the number of the result into the digit at position *i* in the result. For example, when adding *7 + 6*, store *3* in

Compute the end digit of the carry by shifting it using the

Add the carry to the digit at position *i* in the long value a for numbers when the length of *a* is greater than For other cases, the second loop is not entered.

Resave the carry of the result into the digit at position *i* in the result for only the long numbers where the value of *a* is greater

than the value of

Compute the end digit of the carry by shifting it using the

The last digit in the sum is equal to the carry of the entire operation.

The following code block demonstrates the implementation of the multiplication operation:

Objects/longobject.c (line no 3151)

```
/* Grade school multiplication, ignoring the signs.
* Returns the absolute value of the product, or NULL if error.
*/ // -> 1
static PyLongObject* x_mul(PyLongObject *a, PyLongObject *b)
{
PyLongObject *z;
...
z = _PyLong_New(size_a + size_b); // -> 2
...
memset(z->ob_digit, 0, Py_SIZE(z) * sizeof(digit)); // -> 3
...
else {          /* a is not the same as b -- gradeschool int mult */
for (i = 0; i < size_a; ++i) {
twodigits carry = 0; // -> 4
twodigits f = a->ob_digit[i]; // -> 5
digit *pz = z->ob_digit + i;
digit *pb = b->ob_digit;
digit *pbend = b->ob_digit + size_b;
```

```
while (pb < pbend) {
carry += *pz + *pb++ * f; // -> 6
*pz++ = (digit)(carry & PyLong_MASK); // -> 7
carry >>= PyLong_SHIFT; // > 8
...
return long_normalize(z); // -> 9
}
```

Code insights are as

One of the many hilarious comments in the Python source code tree that basically explains the implementation to be similar to high school mathematics.

Create a new variable and assign memory for the variable result.

Set the value in all bytes to the value zero.

Initialize the carry to the value zero.

Set the value of *f* to the value at digit

Multiply the digit at location *i* with the nonvarying digit in the iteration, similar to high school mathematics.

Compute the carry at the current stage and save it in the result pointed to by variable

Compute the carry by shifting it using

Return the multiplied value of the two long values.

Bitwise operations

The bitwise operations of AND and OR perform individual bitwise operations on the digits in the **long** object. The implementation of the bitwise operations is all implemented within a single function, and the logic is more or less the same.

The following code block demonstrates the implementation of the bitwise operations:

```
Objects/longobject.c (line no 4837)
static PyObject* long_and(PyObject *a, PyObject *b)
{
PyObject *c;
CHECK_BINOP(a, b);
c = long_bitwise((PyLongObject*)a, '&', (PyLongObject*)b); // -> 1
return c;
}

static PyObject* long_xor(PyObject *a, PyObject *b)
{
PyObject *c;
CHECK_BINOP(a, b);
c = long_bitwise((PyLongObject*)a, '^', (PyLongObject*)b); // -> 2
return c;
}
```

```
static PyObject* long_or(PyObject *a, PyObject *b)
{
PyObject *c;
CHECK_BINOP(a, b);

c = long_bitwise((PyLongObject*)a, '|', (PyLongObject*)b); // -> 3
return c;
}

static PyObject* long_bitwise(PyLongObject *a, char op,  /* '&', '|',
'^' */ PyLongObject *b)
{
int nega, negb, negz;
Py_ssize_t size_a, size_b, size_z, i;
PyLongObject *z;
...
/* We allow an extra digit if z is negative, to make sure that
the final two's complement of z doesn't overflow. */
z = _PyLong_New(size_z + negz); // -> 4
...
/* Compute digits for overlap of a and b. */
switch(op) {
case '&':
for (i = 0; i < size_b; ++i)
z->ob_digit[i] = a->ob_digit[i] & b->ob_digit[i]; // -> 5
break;
case '|':
for (i = 0; i < size_b; ++i)
z->ob_digit[i] = a->ob_digit[i] | b->ob_digit[i]; // -> 6
break;
case '^':
for (i = 0; i < size_b; ++i)
```

```
z->ob_digit[i] = a->ob_digit[i] ^ b->ob_digit[i]; // -> 7
break;
default:

Py_UNREACHABLE();
}
/* Copy any remaining digits of a, inverting if necessary. */
if (op == '^' && negb)
for (; i < size_z; ++i)
z->ob_digit[i] = a->ob_digit[i] ^ PyLong_MASK; // -> 8
else if (i < size_z)
memcpy(&z->ob_digit[i], &a->ob_digit[i],  (size_z-i)*sizeof(digit)); // -
> 9
...
return (PyObject *)maybe_small_long(long_normalize(z)); // -> 10
}
```

Code insights are as

The **long_bitwise** generic function is called with the operator **&** to perform the logical AND operation.

The **long_bitwise** generic function is called with the operator **|** to perform the logical OR operation.

The **long_bitwise** generic function is called with the operator ^ to perform the logical XOR operation.

Allocate a new **long** variable to store the result of the operation.

Perform a bitwise AND operation for the digits overlapping between the two operands.

Perform a bitwise OR operation for the digits overlapping between the two operands.

Perform a bitwise XOR operation for the digits overlapping between the two operands.

For bitwise **&** and **|** operations, the result does not depend on the nonoverlapping results and hence only for the ^ operation the digits in the variable a is considered and masked with

For the other operations, copy the remaining digits from *a* to the result.

Return the value of the bitwise operation.

The Float object

The **float** object stores the representation of real values used in both application and scientific computations where the applications of Python are very prevalent. This section demonstrates the structure, creation, and implementation of some arithmetic operations, which are possible on the type.

The following code block demonstrates the storage of the **float** object:

Include/floatobject.c (line no 15)

```
typedef struct {
PyObject_HEAD
double ob_fval; // -> 1
} PyFloatObject;
```

Code insight is as

The value for a **float** object is stored as a C **double** type.

The type object

The type of the **float** object contains the possible operations, construction function, deletion function, which are possible on the object:

```
PyTypeObject PyFloat_Type = {
PyVarObject_HEAD_INIT(&PyType_Type, 0)
"float", // -> 1
sizeof(PyFloatObject), // -> 2
(destructor)float_dealloc,                  /* tp_dealloc */ // -> 3
...
(reprfunc)float_repr,                       /* tp_repr */ // -> 4
...
(hashfunc)float_hash,                       /* tp_hash */ // -> 5
...
float_richcompare,                          /* tp_richcompare */ // -
> 6
...
float_new,                                  /* tp_new */ // -> 7
};
```

Code insights are as

The string representation of the type of object.

The size of the **float** object is equal to the size of the structure.

The **destructor** function for the **float** type.

The representation function to convert a **float** object to a string.

The function to hash a **float** value.

The function to compare two **float** values.

The function for creating a new **float** object.

Creating a new float object

The creation of a new floating object includes allocating memory to the double value in the

The following code block demonstrates the implementation of memory allocation for the **float** object:

Objects/floatobject.c (line no 1616)

```
static PyObject* float_subtype_new(PyTypeObject *type, PyObject
*x)
{
PyObject *tmp, *newobj;

assert(PyType_IsSubtype(type, &PyFloat_Type));
tmp = float_new_impl(&PyFloat_Type, x); // -> 1
...
newobj = type->tp_alloc(type, 0); // -> 2
...
((PyFloatObject *)newobj)->ob_fval = ((PyFloatObject *)tmp)-
>ob_fval; // -> 3
Py_DECREF(tmp); // -> 4
return newobj;
}
```

Code insights are as

Create a temporary **float** object. The temporary **float** object has been created to handle internal operations such as conversion from the convertible types such as string/long.

Assign memory to the actual **float** object.

Assign the value to the actual **float** object for programmatic usage and decrement the reference to the temporary object.

// float_new_impl internally calls a number of functions and finally calls
// PyNumber_Float the internal calls are abstracted away for simplicity.

Objects/abstract.c (line no 1450)

```
PyObject* PyNumber_Float(PyObject *o)
{
PyNumberMethods *m;
...
m = o->ob_type->tp_as_number;
if (m && m->nb_float) {/* This should include subclasses of float
*/
...
return PyFloat_FromDouble(val); // -> 1
}
if (m && m->nb_index) {
PyObject *res = PyNumber_Index(o);
```

```
if (!res) {
return NULL;
}
double val = PyLong_AsDouble(res); // -> 2
Py_DECREF(res);
...
return PyFloat_FromDouble(val); // -> 3
}
...
return PyFloat_FromString(o); // -> 4
}
```

Code insights are as

For all classes and subclasses of the **float** object, convert from **float** to **double** using the **PyFloat_FromDouble** function.

For all classes of the **long** object, convert to **double** using the **PyLong_AsDouble** function and later convert to **float** using the

The function to convert a C **double** value to

If the type of the input value is a string, use the **PyFloat_FromString** function to convert a **float** into a string.

Arithmetic operations

The arithmetic operations on **double** type operate on the C **double** value within each of the objects. The addition and subtraction operators are explained as follows, whereas the other operations are left to the curiosity of the readers:

Objects/floatobject.c (line no 543)

```
static PyObject* float_add(PyObject *v, PyObject *w)
{
double a,b;
...
a = a + b; // -> 1
...
return PyFloat_FromDouble(a); // -> 2
}
```

Objects/floatobject.c (line no 555)

```
static PyObject* float_sub(PyObject *v, PyObject *w)
{
double a,b;
...
a = a - b; // -> 3
...
return PyFloat_FromDouble(a); // -> 4
}
```

Code insights are as

The return value for the sum of two numbers is equal to the added value of the **double** values within each

Convert the **double** value into float and return the

The return value for the difference of two numbers is equal to the subtracted value of the **double** values within each

Convert the **double** value into **float** and return the

Comparison operations

The function to compare two floating-point objects checks the internal value stored as a

The following code block demonstrates the implementation of the comparison operation:

```
static PyObject* float_richcompare(PyObject *v, PyObject *w, int
op)
{
double i, j;
int r = 0;
...
Compare:
PyFPE_START_PROTECT("richcompare", return NULL)
switch (op) {
case Py_EQ:
r = i == j; // -> 1
break;
case Py_NE:
r = i != j; // -> 2
break;
case Py_LE:
r = i <= j; // -> 3
break;
case Py_GE:
r = i >= j; // -> 4
```

```
break;
case Py_LT:
r = i < j; // -> 5
break;

case Py_GT:
r = i > j; // -> 6
break;
}
PyFPE_END_PROTECT(r)
return PyBool_FromLong(r);
...
}
```

Code insights are as

When the comparison is equal to check for the equality of the **double** values.

When the comparison is not equal to check for the equality of the **double** values.

When the comparison is less than equal to check for the comparison of the **double** values using **<=** operator.

When the comparison is greater than equal to check for the comparison of the **double** values using **>=** operator.

When the comparison is lesser than check for the comparison of the **double** values using **<** operator.

When the comparison is lesser than check for the comparison of the **double** values using > operator.

The None object

Every programming language has a representation for types that do not hold any valid value. In C/C++, they are popularly known as null or the value of a pointer that does not hold any reference to memory. Incorrectly, handling these references is known to create many issues at runtime leading to unpredictable program behavior. In Python, the type is referred to as **None** and refers to all variables that do not contain a defined value.

The following code block demonstrates the structure of the **None** object:

Objects/object.c (line no 1682)

```
PyObject _Py_NoneStruct = {// -> 1
_PyObject_EXTRA_INIT
1,   // -> // -> 3
};
```

Code insights are as follows:

The **_Py_NoneStruct** is an instance of the generic

The reference count of the object is initialized to

The **type** object contains the name, size, and valid operations that can be performed on the instance of the object.

The none type

The **type** object encompasses the name, size, and operations that can be performed on an instance of the type.

The following code block demonstrates the structure of the **Null** type object:

Objects/object.c (line no 1682)

```
PyTypeObject _PyNone_Type = {
"NoneType", // -> 1
...
none_dealloc,          /*tp_dealloc*/ /*never called*/ // -> 2
...
none_repr,             /*tp_repr*/ // -> 3
&none_as_number,       /*tp_as_number*/ // -> 4
...
none_new,              /*tp_new */ // -> 5
};
```

Code insights are as

The name of the type is set as the string

The deallocation function for the type. The function is not called as the **NoneType** cannot be deallocated during the program lifecycle.

The function to convert the type to a string representation.

Numerical operations can be performed on the type. For **NoneType** objects, the only numerical operation supported is the conversion to a Boolean value.

Function to allocate a new instance of the type. For **NoneType** objects, the object

Creation of the none object

A Python program has one singleton instance of the **None** object. Any new required references increment reference count and access the same object. The following code block demonstrated shows the usage of the singleton instance of the **None** object:

Objects/object.c (line no 1588)

```
static PyObject * none_new(PyTypeObject *type, PyObject *args,
PyObject *kwargs)
{
if (PyTuple_GET_SIZE(args) || (kwargs &&
PyDict_GET_SIZE(kwargs))) {
PyErr_SetString(PyExc_TypeError, "NoneType takes no arguments");
return NULL;
}
Py_RETURN_NONE; // -> 1
}
```

```
return Py_INCREF(Py_None), Py_None // -> 2
```

Code insights are as

Return a reference to the singleton **Py_None** structure.

Increment the reference to the **Py_None** structure and return the reference.

Operations on the none object

The operations on the **None** object are very few in number and only support the conversion to a Boolean value, the code of which is demonstrated as follows.

The following code block demonstrates the implementation of the Boolean representation of the **None** object:

Objects/object.c (line no 1598)

```
static int none_bool(PyObject *v)
{
return 0; // -> 1
}
```

Code insight is as follows:

The Boolean representation of **None** type being false always the value **0** is returned.

Representation of the none object

The string representation of the **None** object returns the static string

Objects/object.c (line no 1571)

```
static PyObject* none_repr(PyObject *op)
{
return PyUnicode_FromString("None"); // -> 1
}
```

Code insights is as

The string representation of **None** type returns the Unicode string

Conclusion

This chapter covered the internals of basic data types in Python, which are the integer, floating-point numbers, and the Boolean value. The Boolean type has only two possible values, which are **True** and **False,** each stored as an object of the type The type of the Boolean object denotes the size, name, and the operations possible on the type.

The **long** object stores the value internally as an array of digits. The arithmetic and bitwise operations on the **long** object were covered in depth. The memory allocation to the **long** object includes the allocation of memory to all digits.

The floating-point number internally stores the value as a C **double** variable and performs operations using basic mathematical operators, unlike the **long** type, which operates on the individual digits.

The **None** type is a single **PyObject,** the value of which is referenced at multiple points in the program by incrementing the reference count. The only numerical operation supported on the **NoneType** is the conversion to a Boolean value.

The upcoming chapter will cover the internals of the structure and operations on the Python iterable objects, which are the lists and tuples.

CHAPTER 3

Iterable Sequence Objects

In the previous chapter, we covered the working of the basic Python types, that is, integers, floating numbers, Boolean, and so on. Although basic types take care of the atomic operations, containers are useful to store multiple values of the same/different types.

Iterable objects such as arrays and dictionaries are the basic data structures used from beginners to the most advanced programmers. Each data structure is optimized for a particular set of use cases and outperforms the others for the operation. For example, **sets** outperform lists for searching elements in a collection. Dictionaries are highly optimized for searching key-value pairs. This chapter covers the internals of lists and tuples by describing the structure, type, and operations of these data structures.

Structure

In this chapter, we will cover the following topics:

The list object

The list type

Creating a list

Accessing an element in a list

Assigning an element in a list

Fetching the length of a list

Removing an element from the list

Freeing all the elements in the list

Checking an element in a list

List iteration

Fetching the iterator

Iterating the elements in the list

Tuples

The tuple type

Creation of the tuple object

Hashing of the tuple object

Unpacking the elements in a tuple object

Objective

After studying this unit, you will be able to understand the structure and working of the operations on a Python list. You will also understand the structure and working of the operations on a Python tuple.

The list object

Understanding the data storage within the list object explains the heterogeneity of the data structure.

The following code block demonstrates the structure of the list object:

Include/listobject.h (Line no 23)

```
typedef struct {
PyObject_VAR_HEAD // -> 1
PyObject **ob_item; // -> 2
...
Py_ssize_t allocated; // -> 3
} PyListObject;
```

Code insights are as

The **list** object is a variable Python object that contains the

Pointer to the list of items, each of them is a Since each of them is stored as a pointer to a they can be any valid Python type explaining the heterogeneity of the data structure.

A number of allocated items in the list.

The list type

The type of the **list** object describes the name, size, and all valid operations that can be performed on it:

Objects/listobject.c (Line no 3030)

```
PyTypeObject PyList_Type = {
PyVarObject_HEAD_INIT(&PyType_Type, 0)
"list", // -> 1
sizeof(PyListObject), // -> 2
0,
(destructor)list_dealloc,                   /* tp_dealloc */ // -> 3
...
(reprfunc)list_repr,                        /* tp_repr */ // -> 4
0,                                          /* tp_as_number */
&list_as_sequence,                          /* tp_as_sequence */ // ->
5
&list_as_mapping,                           /* tp_as_mapping */ // ->
6
PyObject_HashNotImplemented,                /* tp_hash */ // -> 7
...
PyObject_GenericGetAttr,                    /* tp_getattro */ // -> 8
...
(traverseproc)list_traverse,                /* tp_traverse */ // -> 9
(inquiry)_list_clear,                       /* tp_clear */ // -> 10
list_richcompare,                           /* tp_richcompare */ // ->
11
```

```
0,                                      /* tp_weaklistoffset */
list_iter,                              /* tp_iter */ // -> 12
0,                                      /* tp_iternext */
list_methods,                           /* tp_methods */ // -> 13
...
PyObject_GC_Del,                        /* tp_free */ // -> 14
};
```

Code insights are as

The type of the object is described as This is obtained when we use the type built-in function on a list object.

The size of the list object is defined by the size of the

The deallocator of the **list** object.

The function creates a string representation of the **list** object when output to a file descriptor.

The sequence methods on the list will be covered in the subsequent sections.

The mapping methods on the list will be covered in the subsequent sections.

PyObject_HashNotImplemented indicates that the hashing is not implemented by the object as it has variable data.

PyObject_GenericGetAttr indicates the usage of the generic function for getting attributes from the tuple object. It is suggested to explore the implementation of this function and is beyond the scope of this book.

The **list_traverse** method is used to traverse all the elements in the list.

The **clear** method is used to clear all the elements in the list.

Method to compare two lists for equality.

Method to fetch the iterator to traverse the elements of the list.

Fetch all the methods in the list object such as and so on.

Method to clear the memory allocated to the **list** object.

Creating a list

Creating a list involves allocating memory to all the object pointers held by it. As seen in the preceding section on *The List Object* a list is an array of pointers to **PyObjects,** and each of which can be of a different type enabling heterogeneity of the data structure. In compiled languages such as C/Java, a list/array can only hold data of one type.

The following code sample demonstrates the generation of an **opcode** for a program creating a list:

```
a = [1, 2, 3]
```

Opcode

```
1      0 LOAD_CONST               0 (1)
2  LOAD_CONST                     1 (2)
4  LOAD_CONST                     2 (3)
6  BUILD_LIST                  3  // -> 1
8  STORE_NAME                     0 (l)
10  LOAD_CONST                    3 (None)
12  RETURN_VALUE
```

The code insight is as

The creation of a list uses the opcode **BUILD_LIST,** which creates a **PyListObject** object, allocates memory for it, and adds the data,

that is, **3,** into it. The implementation is demonstrated as follows.

The following code block demonstrates the implementation of the **BUILD_LIST** opcode:

Python/ceval.c (Line no 2689)

```
case TARGET(BUILD_LIST): {
PyObject *list =  PyList_New(oparg); // -> 1
...

while (--oparg >= 0) {
PyObject *item = POP(); // -> 2
PyList_SET_ITEM(list, oparg, item); // -> 3
}
...
}
```

The code insights are as

Allocate memory to the list object using the **PyList_New** constructor.

The argument to the opcode indicates the number of elements to be added to the list during creation. These are added to the function stack using the **LOAD_CONST** opcode. The **POP** opcode pops it from the stack.

The **PyList_SET_ITEM** function adds the item into the list in the reverse order, that is, element **3** is added at index followed by **2**

at index and so on. This is because the elements are present on the stack in the reverse order when added using the **LOAD_CONST** opcode.

Accessing an element in a list

Accessing an element in a list involves returning a pointer to the object at the requested index or throwing an exception when an invalid index has been requested.

The code sample is as follows:

```
a = [1, 2, 3]
print(a[1])
```

Opcode

```
...
2           10 LOAD_NAME                1 (print)
12 LOAD_NAME                 0 (a)
14 LOAD_CONST                0 (1)
16 BINARY_SUBSCR             // -> 1
18 CALL_FUNCTION             1
20 POP_TOP
...
```

The code insight is as

The compiler creates the **BINARY_SUBSCR** opcode to indicate the program logic accessing an element of a list using its index.

The following code block demonstrates the implementation of the **BINARY_SUBSCR** opcode:

Python/ceval.c (Line no 1581)

```
case TARGET(BINARY_SUBSCR): {
PyObject *sub = POP(); // -> 1
PyObject *container = TOP(); // -> 2
PyObject *res = PyObject_GetItem(container, sub); // -> 3
...
SET_TOP(res); // -> 4

...
}
```

Objects/abstract.c (Line no 143)

```
PyObject * PyObject_GetItem(PyObject *o, PyObject *key)
{
PyMappingMethods *m;
PySequenceMethods *ms;

...
ms = o->ob_type->tp_as_sequence; // -> 5

if (ms && ms->sq_item) {
if (PyIndex_Check(key)) {
...
return PySequence_GetItem(o, key_value); // -> 6
}
...
```

```
}
```

Objects/abstract.c (Line no 1680)

```
PyObject * PySequence_GetItem(PyObject *s, Py_ssize_t i)
{
...
return m->sq_item(s, i); // -> 7
...
}
```

Objects/listobject.c (line no 461)

```
static PyObject* list_item(PyListObject *a, Py_ssize_t i)
{
if (!valid_index(i, Py_SIZE(a))) {
if (indexerr == NULL) {
indexerr = PyUnicode_FromString(

"list index out of range"); // -> 8
if (indexerr == NULL)
return NULL;
}
...
}
...
return a->ob_item[i]; // -> 9
}
```

Code insights are as

Fetch the index to be assigned from the list present at the top of the stack.

Fetch the list from which the element has to be assigned.

Fetch the element from the list using the **PyObject_GetItem** function.

Set the obtained result on top of the stack.

Fetch the sequence methods from the stack.

Fetch the item using the stack.

Call the **sq_item** function on the list type to which internally calls the **list_item** function.

Raise the list index out of range exception if the index is greater than the length of the list.

Fetch the item from the list from the **ob_item** array.

Assigning an element in a list

The previous section covered accessing an element in a list by using the index. Another common operation performed on lists is assigning an element to a particular index.

Code sample is as follows:

```
a = [1, 2, 3]
a[1] = 3
```

Opcode

```
...
2           10 LOAD_CONST               2 (3)
12 LOAD_NAME                0 (a)
14 LOAD_CONST               0 (1)
16 STORE_SUBSCR             // -> 1
18 LOAD_CONST               3 (None)
20 RETURN_VALUE
```

The code insight is as

The **STORE_SUBSCR** opcode is used to assign a particular value at a particular index in the list.

The following code block demonstrates the implementation of the **STORE_SUBSCR** opcode:

Python/ceval.c (Line no 1840)

```
case TARGET(STORE_SUBSCR): {
PyObject *sub = TOP(); // -> 1
PyObject *container = SECOND(); // -> 2
PyObject *v = THIRD(); // -> 3
...
/* container[sub] = v */
err = PyObject_SetItem(container, sub, v); // -> 4
...

}
```

Objects/abstract.c (Line no 190)

```
int PyObject_SetItem(PyObject *o, PyObject *key, PyObject *value)
{
...
if (o->ob_type->tp_as_sequence) {
if (PyIndex_Check(key)) {
...
return PySequence_SetItem(o, key_value, value); // -> 5
}
...
}
}
...
return -1;
```

```
}
```

Objects/abstract.c (Line no 1734)

```
int PySequence_SetItem(PyObject *s, Py_ssize_t i, PyObject *o)
{
....
return m->sq_ass_item(s, i, o); // -> 6
}
...
}
```

Objects/listobject.c (Line no 788)

```
static int list_ass_item(PyListObject *a, Py_ssize_t i, PyObject *v)
{

...
Py_SETREF(a->ob_item[i], v); // -> 7
return 0;
}
```

Code insights are as

Fetch the index to be assigned from the list present at the top of the stack.

Fetch the list from which the element has to be assigned.

Fetch the value to be inserted at the location in the array.

The **PyObject_SetItem** function is used to assign the element from the container not just for lists but also generic container types.

The **PyObject_SetItem** function internally calls the **PySequence_SetItem** for sequence types such as lists.

The implementation of this feature belongs to the **sq_ass_item** prototype, which has been covered in *Chapter 1 on Generic Python*

The **Py_SETREF** macro sets the **PyObject** to the new value passed by the user.

Fetching the length of a list

Many operations in a program depend on the length of the list. This section covers the implementation of the length of a list.

Code sample is as follows:

```
a = [1, 2, 3]
len(a)
```

Opcode

```
...
LOAD_NAME                1 (len) // -> 1
12 LOAD_NAME             0 (a)
14 CALL_FUNCTION         1
16 POP_TOP
18 LOAD_CONST            3 (None)
```

Code insight is as

The built-in function **len** is the most common way to fetch the length of the list or any iterable types in general.

Python/clinic/bltinmodule.c (line no 564)

```
#define BUILTIN_LEN_METHODDEF    \
```

{"len", (PyCFunction)builtin_len, METH_O, builtin_len__doc__}, // -> 1

Python/clinic/bltinmodule.c (line no 1546)

```
static PyObject * builtin_len(PyObject *module, PyObject *obj)
{
...
res = PyObject_Size(obj); // -> 2
...
}
```

Objects/abstract.c (line no 46)

```
Py_ssize_t PyObject_Size(PyObject *o)
{
PySequenceMethods *m;

m = o->ob_type->tp_as_sequence;
if (m && m->sq_length) {
Py_ssize_t len = m->sq_length(o); // -> 3
return len;
}
...
}
```

Objects/listobject.c (line no 439)

```
static Py_ssize_t list_length(PyListObject *a)
{
```

return Py_SIZE(a); // -> 4
}
Include/object.h (line no 123)

#define Py_SIZE(ob) (_PyVarObject_CAST(ob)->ob_size) // -> 5

Code insights are as

The built-in **len** method is declared in the **bltinmodule.h** file with the documentation.

The function internally calls

PyObject_Size internally calls the **sq_length** implementation for the container type.

The implementation of the **sq_length** internally calls the **Py_SIZE** macro standard to all variable Python types.

The macro returns the value for the **ob_size** attribute for the variable type.

Removing an element from the list

This section covers removing an element from a list by index. This operation is usually a complex one as it involves completely rearranging the elements of the list after the removal.

Code sample is as follows:

```
a = [1, 2, 3]
del a[0]
```

Opcode

```
2           10 LOAD_NAME                0 (a)
12 LOAD_CONST                3 (0)
14 DELETE_SUBSCR             // -> 1
16 LOAD_CONST                4 (None)
18 RETURN_VALUE
```

Code insight is as

The **DELETE_SUBSCR** opcode is used to remove an element from the list using the index.

The following code block demonstrates the implementation of the **DELETE_SUBSCR** opcode:

Python/ceval.c (line no 1855)

```
case TARGET(DELETE_SUBSCR): {
PyObject *sub = TOP(); // -> 1
PyObject *container = SECOND(); // -> 2
...
/* del container[sub] */
err = PyObject_DelItem(container, sub); // -> 3
...
}
```

Objects/abstract.c (line no 222)

```
int PyObject_DelItem(PyObject *o, PyObject *key)

{
...
if (o->ob_type->tp_as_sequence) {
if (PyIndex_Check(key)) {
...
return PySequence_DelItem(o, key_value); // -> 4
...
}
}
```

Objects/abstract.c (line no 222)

```
int PySequence_DelItem(PyObject *s, Py_ssize_t i)
{
...
return m->sq_ass_item(s, i, (PyObject *)NULL); // -> 5
...
```

```
}
```

Objects/listobject.c (line no 789)

```
static int list_ass_item(PyListObject *a, Py_ssize_t i, PyObject *v)
{
...
if (v == NULL)
return list_ass_slice(a, i, i+1, v); // -> 6
...
```

Code insights are as

Fetch the index to be removed from the list present at the top of the stack.

Fetch the list from which the element has to be removed.

Call the generic **PyObject_DelItem** function to remove an element from a container.

The **PyObject_DelItem** internally calls the **PySequence_DelItem** function to remove the element from sequence types.

The **sq_ass_item** function prototype handles splitting the list into the respective parts by index.

When the value to be removed is passed as the list calls the **slice** method with the **NULL** value, which internally uses ***memmove*** to create the split list. The function internally creates a new array and uses the standard library function to reduce the time required to copy the elements from the source array. Covering the implementation of the slicing method is beyond the scope of this book.

Freeing all the elements in the list

Deleting a list includes decrementing the references to all elements in the list and removing the allocated memory to the container.

The code sample is as follows:

```
a = [1, 2, 3]
del a
```

Opcode

```
2           10 DELETE_NAME               0 (a) // -> 1
12 LOAD_CONST                3 (None)
14 RETURN_VALUE
```

Code insight is as

The ***DELETE_NAME*** opcode is used to remove all the elements from the list along with the allocated memory to the container.

The following code block demonstrating the implementation of the **DELETE_NAME** opcode:

Python/ceval.c (line no 2301)

```
case TARGET(DELETE_NAME): {
```

```
PyObject *name = GETITEM(names, oparg); // -> 1
PyObject *ns = f->f_locals; // -> 2
...
err = PyObject_DelItem(ns, name); // -> 3
...
}
```

Objects/abstract.c (line no 223)

```
int PyObject_DelItem(PyObject *o, PyObject *key)
{
...

if (m && m->mp_ass_subscript)
return m->mp_ass_subscript(o, key, (PyObject*)NULL); // -> 4
}
```

Objects/listobject.c (line no 360)

```
static void list_dealloc(PyListObject *op)
{
...
if (op->ob_item != NULL) {
...
i = Py_SIZE(op);
while (--i >= 0) {
Py_XDECREF(op->ob_item[i]); // -> 5
}
PyMem_FREE(op->ob_item); // -> 6
}
...
```

Py_TYPE(op)->tp_free((PyObject *)op); // -> 7
Py_TRASHCAN_END
}
PyObject_GC_Del, /* tp_free */ // -> 8

Code insights are as

Get the name to be deleted in the current case it will be the string

Get the dictionary containing all local variables in the function frame scope.

Call the **PyObject_DelItem** generic function, which deletes an element from the dictionary using the name.

As the local variables are indexed by name in a dictionary, it uses the **mp_ass_subscript** function prototype for containers implementing the mapping. Tracing the entire flow of this function call is complex, and it internally calls the **list_dealloc** function.

Decrement the reference of every element in the list.

Free the memory allocated to the items array in the list. The items array holds all the pointers to the **PyObjects** in the list.

Call the **free** method on the list, which functions as the destructor for the container type.

Free the memory allocated using Python's internal memory allocator.

Checking an element in a list

Checking if an element exists in a list of objects is a common operation used in many user applications. Most programs accept an array of input values and check if the elements belong to an acceptable dataset.

The code sample is as follows:

```
a = [1, 2, 3]
1 in a
```

Opcode

```
2          10 LOAD_CONST               0 (1)
12 LOAD_NAME                0 (a)
14 COMPARE_OP               6 (in) // -> 1
16 POP_TOP
18 LOAD_CONST               3 (None)
20 RETURN_VALUE
```

Code insight is as

The **COMPARE_OP** opcode is used to check if an element exists in a list of objects.

The following code block demonstrates the implementation of the **COMPARE_OP** opcode:

Python/ceval.c (line no 2974)

```
case TARGET(COMPARE_OP): {
PyObject *right = POP(); // -> 1
PyObject *left = TOP(); // -> 2
PyObject *res = cmp_outcome(tstate, oparg, left, right); // -> 3
...
}
```

Python/ceval.c (line no 5065)

```
static PyObject * cmp_outcome(PyThreadState *tstate, int op,
PyObject *v, PyObject *w) {
int res = 0;

switch (op) {
...
case PyCmp_IN: // -> 4
res = PySequence_Contains(w, v); // -> 5
if (res < 0)
return NULL;
...
}
}
```

Objects/abstract.c (line no 2082)

```
int PySequence_Contains(PyObject *seq, PyObject *ob)
{
...
if (sqm != NULL && sqm->sq_contains != NULL)
return (*sqm->sq_contains)(seq, ob); // -> 6
...
}
```

Objects/listobject.c (line no 446)

```
static int list_contains(PyListObject *a, PyObject *el)
{
...
for (i = 0, cmp = 0 ; cmp == 0 && i < Py_SIZE(a); ++i) {
item = PyList_GET_ITEM(a, i); // -> 7
cmp = PyObject_RichCompareBool(el, item, Py_EQ); // -> 8
...
}

return cmp;
}
```

Code insights are as

Fetch the element at the RHS of the operand.

Fetch the element at the LHS of the operand.

The **cmp_outcome** function compares to check if the element exists in the list.

The **PyCmp_IN** operator denotes the operation being performed is checking for the existence of the element.

The **PySequence_Contains** function checks for the element in the list/any sequence type such as tuple.

The function internally calls the **sq_contains** function prototype for checking if the element exists in the list.

Fetch the item at the location during iterating through the elements in the list.

Check if the element is equal to the element being checked and return the result of the comparison.

List iteration

Iterating through the elements of the list involves traversing the elements and returning the element at each location.

Code sample is as follows:

```
a = [1, 2, 3]
for ele in a:
print(ele)
```

Opcode

```
3           10 LOAD_NAME                0 (a)
12 GET_ITER                         // -> 1
>>    14 FOR_ITER                   12 (to 28) // -> 2
16 STORE_NAME                       1 (ele)
4           18 LOAD_NAME                2 (print)
20 LOAD_NAME                        1 (ele)
22 CALL_FUNCTION                    1
24 POP_TOP
26 JUMP_ABSOLUTE                    14 // -> 3
>>    28 LOAD_CONST                 3 (None)
30 RETURN_VALUE
```

Code insights are as

The **GET_ITER** opcode is used to fetch the iterator object to traverse the elements in the list.

The **FOR_ITER** opcode fetches the next element to be visited in the iterable object; otherwise, it throws a **StopIteration** exception once the completion of the iteration of all elements.

Once the current iteration code is completed, jump back to opcode **14** for the next cycle of iteration.

Fetching the iterator

The following code demonstrates the implementation of the **GET_ITER** opcode:

Python/ceval.c (line no 3156)

```
case TARGET(GET_ITER): {
PyObject *iterable = TOP(); // -> 1
PyObject *iter = PyObject_GetIter(iterable); // -> 2
...
SET_TOP(iter); // -> 3
...
}
```

Objects/abstract.c (line no 2569)

```
PyObject * PyObject_GetIter(PyObject *o)
{
...
if (PySequence_Check(o))
return PySeqIter_New(o); // -> 4
...
}
```

Objects/abstract.c (line no 2569)

```
PyObject* PySeqIter_New(PyObject *seq)
```

```
{
...
it = PyObject_GC_New(seqiterobject, &PySeqIter_Type); // -> 5
...
}
```

Objects/abstract.c (line no 2569)

```
typedef struct {
PyObject_HEAD

Py_ssize_t it_index; // -> 6
PyObject *it_seq; /* Set to NULL when iterator is exhausted */ //
-> 7
} seqiterobject;
```

Code insights are as

Fetch the list to be iterated from the top of the stack.

Construct the iterator object for the list.

Set the iterator to the top of the function stack.

Create and fetch the iterator for the sequence using the **PySeqIter_New** function.

Create the iterator object for the list.

The current index is currently traversed by the iterator.

Pointer to the iterable object to be iterated.

Iterating the elements in the list

The following code block demonstrates the implementation of the **FOR_ITER** opcode:

Python/ceval.c (line no 3198)

```
case TARGET(FOR_ITER): {
...
PyObject *next = (*iter->ob_type->tp_iternext)(iter); // -> 1

if (next != NULL) {
PUSH(next); // -> 2
...
}

if (_PyErr_Occurred(tstate)) {
if (!_PyErr_ExceptionMatches(tstate, PyExc_StopIteration)) {
goto error;
} // -> 3
...
}
...
}
```

Objects/listobject.c (line no 3168)

```
static PyObject *
```

```
listiter_next(listiterobject *it)
{
...
if (it->it_index < PyList_GET_SIZE(seq)) {// -> 4
item = PyList_GET_ITEM(seq, it->it_index); // -> 5
++it->it_index; // -> 6
Py_INCREF(item);

return item; // -> 7
}

it->it_seq = NULL; // -> 8
Py_DECREF(seq); // -> 9
return NULL; // -> 10
}
```

Code insights are as

Fetch the next element using the **tp_iternext** method on the iterator object.

Push the result into the function stack pointer.

When the result is not provided, the **PyExc_StopIteration** exception is thrown.

Check if the current index being iterated is lesser than the size of the iterable object.

Fetch the item at the current index pointed by

Increment the iteration for the next element in the list.

Return the item to the current index.

At the end of iteration of all elements in the list, initiate the destruction of the iterator object.

Decrement the reference of the sequence if **o** deallocates the memory.

Return **NULL** when the iteration is completed.

The tuple object

The structure and definitions of the tuple object are very similar to a list. The primary difference being that the elements in a tuple do not vary once created:

Include/cpython/tupleobject.h (line no 15)

```
typedef struct {
PyObject_VAR_HEAD // -> 1
/* ob_item contains space for 'ob_size' elements.
Items must normally not be NULL, except during construction when
the tuple is not yet visible outside the function that builds it. */
PyObject *ob_item[1]; // -> 2
} PyTupleObject;
```

Code insights are as

The tuple object is an object of variable type.

The array of **PyObjects** in the tuple.

The tuple type

The **tuple** type consists of the list of all possible operations that can be performed on elements in the tuple:

Objects/tupleobject.c (line no 829)

```
PyTypeObject PyTuple_Type = {
PyVarObject_HEAD_INIT(&PyType_Type, 0)
"tuple", // -> 1
sizeof(PyTupleObject) - sizeof(PyObject *), // -> 2
sizeof(PyObject *),
(destructor)tupledealloc,                   /* tp_dealloc */ // -> 3
...
(reprfunc)tuplerepr,                        /* tp_repr */ // -> 4
0,                                          /* tp_as_number */
&tuple_as_sequence,                         /* tp_as_sequence */ // ->
5
&tuple_as_mapping,                          /* tp_as_mapping */ // ->
6
(hashfunc)tuplehash,                        /* tp_hash */ // -> 7
...
PyObject_GenericGetAttr,                    /* tp_getattro */ // -> 8
...
(traverseproc)tupletraverse,                /* tp_traverse */ // -> 9
...
tuplerichcompare,                           /* tp_richcompare */ // ->
10
```

```
...
tuple_iter,                             /* tp_iter */ // -> 11

0,                                       /* tp_iternext */
tuple_methods,                          /* tp_methods */ // -> 12
...
tuple_new,                              /* tp_new */ // -> 13
PyObject_GC_Del,                        /* tp_free */ // -> 14
}
```

Code insights are as

The type of the object is described as

The size of the tuple object is defined as the *size of the PyTupleObject - size of the*

The deallocator of the tuple object.

The representation of the tuple object as string to be printed to a file descriptor.

The sequence methods on the tuple will be covered in the subsequent sections.

The mapping methods on the tuple will be covered in the subsequent sections.

The **hash** method indicates that the hashing is implemented by the object as it does not have variable data.

PyObject_GenericGetAttr indicates the usage of the generic function for getting attributes from the **tuple** object. It is suggested to explore the implementation of this function and is beyond the scope of this book.

The **traverseproc** is used to traverse all the elements in the tuple.

The **richcompare** method is used to compare two tuples for the equality of elements.

Method to fetch the iterator to traverse the elements of the tuple.

Fetch all the methods for operations possible on the **tuple** object.

Constructor method for the **tuple** class.

Method to clear the memory allocated to the **tuple** object.

Creation of the tuple object

The previous section covered the creation of a list with three items. In tuples, all the elements to be a part of the data structure must be declared upfront, whereas in lists the elements can be modified post creation.

The code sample is as follows:

```
class a:
pass
t = (a(), a())
```

Opcode

```
1           0 LOAD_BUILD_CLASS
2 LOAD_CONST                0 (object a at 0x103f0eb30, file
"p.py", line 1>)
4 LOAD_CONST                1 ('a')
6 MAKE_FUNCTION             0
8 LOAD_CONST                1 ('a')
10 CALL_FUNCTION            2
12 STORE_NAME               0 (a)
4           14 LOAD_NAME                0 (a)
16 CALL_FUNCTION            0
18 LOAD_NAME                0 (a)
20 CALL_FUNCTION            0
22 BUILD_TUPLE              2 // -> 1
```

```
24 STORE_NAME                1 (t)
26 LOAD_CONST                2 (None)
28 RETURN_VALUE
```

Code insight is as

The **BUILD_TUPLE** opcode is used to create a tuple for dynamic data. If the data is static (such as the compiler optimizes the creation of the tuple to save time at execution. It is suggested to the readers to explore the code flow for static tuples.

The following code block demonstrates the creation of the tuple:

Python/ceval.c (line no 2677)

```
case TARGET(BUILD_TUPLE): {
PyObject *tup = PyTuple_New(oparg); // -> 1
if (tup == NULL)
goto error;
while (--oparg >= 0) {
PyObject *item = POP(); // -> 2
PyTuple_SET_ITEM(tup, oparg, item); // -> 3
}
PUSH(tup);
DISPATCH();
}
```

Objects/tupleobject.c (line no 79)

```
PyObject *
PyTuple_New(Py_ssize_t size)
{
PyTupleObject *op;
...
{
/* Check for overflow */
if ((size_t)size > ((size_t)PY_SSIZE_T_MAX - sizeof(PyTupleObject)
- sizeof(PyObject *)) / sizeof(PyObject *)) {
return PyErr_NoMemory();
}

op = PyObject_GC_NewVar(PyTupleObject, &PyTuple_Type, size); //
-> 4
if (op == NULL)
return NULL;
}
...
return (PyObject *) op;
}
```

Code insights are as

The **PyTuple_New** function allocates memory to the tuple data structure.

Pop the item to be inserted into the tuple from the function stack.

Set the item at the index into the tuple.

Allocate memory from the Python memory allocator for the tuple.

Hashing of the tuple object

Since the data within the tuple object does not vary, it can be hashed as a key for dictionaries. Lists do not implement the hashing function as the data can vary.

The following code block demonstrates the implementation of the hashing function for tuples:

Objects/tupleobject.c (line no 367)

```
static Py_hash_t tuplehash(PyTupleObject *v)
{
Py_ssize_t i, len = Py_SIZE(v);
PyObject **item = v->ob_item;

for (i = 0; i < len; i++) {
Py_uhash_t lane = PyObject_Hash(item[i]); // -> 1
...
acc += lane * _PyHASH_XXPRIME_2;
acc = _PyHASH_XXROTATE(acc);
acc *= _PyHASH_XXPRIME_1; // -> 2
}
/* Add input length, mangled to keep the historical value of
hash(()). */
acc += len ^ (_PyHASH_XXPRIME_5 ^ 3527539UL);
if (acc == (Py_uhash_t)-1) {
```

```
return 1546275796;
}
return acc;
}
```

Code insight is as

Compute the hash of the object at the particular index.

The hash computed is aggregated for all tuple indexes. The implementation of the **hash** function is beyond the scope of this book. The documentation provided by the community at *line no 367* can help interested readers explore further. The point to be demonstrated was the difference between lists and tuples in hashing.

Unpacking the elements in a tuple object

One of the frequent tricks used by developers to return multiple values from a function is to return it as a tuple which is later unpacked as separate values when accepted.

Code sample is as follows:

```
a, b = (1, 2)
1           0 LOAD_CONST               0 ((1, 2))
2 UNPACK_SEQUENCE          2 //   -> 1
4 STORE_NAME               0 (a) // -> 2
6 STORE_NAME               1 (b)
8 LOAD_CONST               1 (None)
10 RETURN_VALUE
```

Code insights are as

The **UNPACK_SEQUENCE** opcode unpacks the data in the tuple and adds them to the function stack.

Once the values are added to the function stack, they are popped out using the **STORE_NAME** opcode and stored in the variable.

The following code block demonstrates the implementation of the **UNPACK_SEQUENCE** opcode:

Python/ceval.c (line no 2320)

```
case TARGET(UNPACK_SEQUENCE): {
...

items = ((PyTupleObject *)seq)->ob_item;
while (oparg--) {
item = items[oparg]; // -> 1
Py_INCREF(item);
PUSH(item); // -> 2
}
}
...
}
```

Code insights are as

Fetch the element at a particular index in the tuple.

Push the item into the function stack. The **STORE_NAME** opcode pops the element from the function stack and stores it in the variable.

Conclusion

This chapter covered the most commonly used data structures in Python that are the list and tuple. The chapter began with the structure of the list, covering the operations possible on the type. Next, the creation of lists, along with operations such as accessing an element at a particular index, deletion of elements was covered.

Deletion of lists includes deallocating each element and later deallocating the memory for the container. List iteration includes two steps, that is, fetching the iterator for iterating and followed by accessing every element. Tuples, although very similar in structure, differ in the way the data structure is hashed.

Unpacking a tuple involves pushing each element to the function stack and later popping every element and storing it to a variable.

The upcoming chapter will discuss the internal workings of the Python mappable types that are sets and dictionaries. Sets and dictionaries are functionally similar and share almost similar structures and implementations.

Reader exercises

Analyze the similarities in operations such as accessing an element by index, iteration between list, and tuple. Although the chapter covers them only for lists, readers are encouraged to research the same for tuples.

CHAPTER 4

Set and Dictionary

In the previous chapter, we covered the workings of the Python iterable types, that is, **lists** and **tuples** . This chapter covers the workings of the hashable types, which are the set and dictionary.

The main purpose of hashable types in programming languages is to provide a consistent search of items in the container types. Dictionaries are key-value containers, where the key can be any hashable value such as a string/integer. Sets are iterable types, which provide *O(1) search* for elements and operations such as AND, OR, union, and intersection.

Structure

In this chapter, we will cover the following topics:

Implementation of the set object

Implementation of the dictionary object

Objective

Understanding the internals of hashable data structures enables the programmers to make informed decisions on choosing them for the right job. For example, **sets** are most optimal over lists for cases where there has to be an element searched within a large number of elements, whereas lists provide ordering of elements useful for many cases such as building stacks and so on. This chapter covers how data structures such as dictionaries/sets help programmers perform *O(1) operation* enabling faster insertion, search, and removal.

The set object

The **set** data structure is very similar to the dictionary in structure and operates almost similar to a dictionary. The difference lies in the part where the key and value remain the same in the dictionary.

Structure of the set object

The **set** object is structurally similar to the dictionary and contains an array that stores the entries in an array of set entries:

Include/setobject.h (line no 24)

```
#define PySet_MINSIZE 8

typedef struct {
PyObject *key; // -> 1
Py_hash_t hash;  // -> Cached hash code of the key */
} setentry;

typedef struct {
PyObject_HEAD

Py_ssize_t fill; // -> 3              /* Number active and dummy entries*/
Py_ssize_t used;  // -> 4              /* Number active entries */
/* The table contains mask + 1 slots, and that's a power of 2.
* We store the mask instead of the size because the mask is more
* frequently needed.
*/
Py_ssize_t mask; // -> 5
/* The table points to a fixed-size smalltable for small tables
```

```
* or to additional malloc'ed memory for bigger tables.
* The table pointer is never NULL which saves us from repeated
* runtime null-tests.
*/
setentry *table; // -> 6

Py_hash_t hash;                 /* Only used by frozenset objects
*/
Py_ssize_t finger;              /* Search finger for pop() */
setentry smalltable[PySet_MINSIZE]; // -> 7
PyObject *weakreflist;          /* List of weak references */
} PySetObject;
```

Code insights are as

The key of the object is also the value in the

The hash of the object in the table stores the set entries.

The number of entries in the table being used in the set, including the dummy entries.

The number of active entries in the table is actually being used.

The mask is the size of the set with a power of

The table stores the entries of the **set** object.

The table is used for faster creation by allowing static allocation of memory for smaller sets of sizes lesser than

Creation of the set object

Creating a set object depends on the size of the set being created. If the size is greater than memory has to be allocated for the set dynamically.

The code sample is as follows:

```
a = {1, 2, 3}
```

Opcode

```
2           0 LOAD_CONST               0 (1)
            2 LOAD_CONST               1 (2)
            4 LOAD_CONST               2 (3)
            6 BUILD_SET                3 // -> 1
            8 STORE_NAME               0 (a)
           10 LOAD_CONST               3 (None)
```

The code insight is as

The creation of a set uses the opcode **BUILD_SET,** which creates a **PySetObject** object, allocates memory for it, and adds the data, that is, **3,** into it. The implementation is demonstrated as follows.

The following code block demonstrates the implementation of the **BUILD_SET** opcode:

Python/ceval.c (Line no 2689)

```
case TARGET(BUILD_SET): {
PyObject *set = PySet_New(NULL); // -> 1
int err = 0;
int i;
if (set == NULL)
goto error;
for (i = oparg; i > 0; i--) {
PyObject *item = PEEK(i); // -> 2

if (err == 0)
err = PySet_Add(set, item); // -> 3
Py_DECREF(item);
}
STACK_SHRINK(oparg); // -> 4
...
}

PyObject* PySet_New(PyObject *iterable) {
return make_new_set(&PySet_Type, iterable); // -> 5
}

static PyObject* make_new_set(PyTypeObject *type, PyObject
*iterable)
{
PySetObject *so;
```

```
so = (PySetObject *)type->tp_alloc(type, 0); // -> 6
if (so == NULL)
return NULL;


so->fill = 0;
so->used = 0; // -> 7
so->mask = PySet_MINSIZE - 1; // -> 8
so->table = so->smalltable; // -> 9
so->hash = -1;
...
return (PyObject *)so;
}
```

Code insights are as

Allocate memory to the set object using the **PySet_New** constructor.

The argument to the opcode indicates the number of elements to be added to the list during creation. These are added to the function stack using the **LOAD_CONST** opcode. The **POP** opcode pops it from the stack.

The **PySet_Add** function adds the item into the set in the reverse order, that is, element **3** is added at index followed by **2** at index and so on. This is because the elements are present on the stack in the reverse order when added using the **LOAD_CONST** opcode.

The function to add elements to the set is covered in the upcoming section.

The **STACK_SHRINK** opcode removes the used elements from the set into the set object.

The **make_new_set** function allocates memory and initializes the set object.

The **tp_alloc** function of the set object assigns memory to the set object.

The used flag is set to **0** indicating there are no elements in the set object.

The size of the set is initialized to a small set.

The **smalltable** is assigned to the table.

Adding an element to a set object

Adding an element to a set object includes adding the element at the value in the array relevant to the hash of the value inserted into the set. It is to be noted that the elements of a set can only be hashable elements such as integers, strings, tuples, and so on. Adding lists or dictionaries to a set can result in an invalid value exception. A set always contains unique values, and adding duplicate values does not change the set in any way. In the previous section, we have covered the creation of a set and how the **PySet_Add** function is used to add an element to a set.

The following code block demonstrates adding values to a set:

Objects/setobject.c (line no 137)

```
static int set_add_entry(PySetObject *so, PyObject *key, Py_hash_t
hash)
{
...
restart:

mask = so->mask;
i = (size_t)hash & mask; // -> 1

entry = &so->table[i]; // -> 2
```

```
if (entry->key == NULL)
goto found_unused; // -> 3

freeslot = NULL;
perturb = hash;

while (1) {// -> 4
if (entry->hash == hash) {// -> 5
PyObject *startkey = entry->key;
/* startkey cannot be a dummy because the dummy hash field is
-1 */

assert(startkey != dummy);
if (startkey == key)
goto found_active; // -> 6
if (PyUnicode_CheckExact(startkey)
&& PyUnicode_CheckExact(key)
&& _PyUnicode_EQ(startkey, key))
goto found_active; // -> 7
table = so->table;
Py_INCREF(startkey);
cmp = PyObject_RichCompareBool(startkey, key, Py_EQ); // -> 8
Py_DECREF(startkey);
if (cmp > 0)                                    /* likely */
goto found_active; // -> 9
...
}
...
if (i + LINEAR_PROBES <= mask) {// -> 10
```

```
for (j = 0 ; j < LINEAR_PROBES ; j++) {
entry++; // -> 11
if (entry->hash == 0 && entry->key == NULL)
goto found_unused_or_dummy; // -> 12
if (entry->hash == hash) {
PyObject *startkey = entry->key;
assert(startkey != dummy);
if (startkey == key)
goto found_active;
if (PyUnicode_CheckExact(startkey)
&& PyUnicode_CheckExact(key)

&& _PyUnicode_EQ(startkey, key))
goto found_active; // -> 12
...
}
else if (entry->hash == -1)
freeslot = entry;
}
}

perturb >>= PERTURB_SHIFT;
i = (i * 5 + 1 + perturb) & mask;

entry = &so->table[i];
if (entry->key == NULL) // -> 13
goto found_unused_or_dummy; // -> 14

}

found_unused_or_dummy:
```

```
if (freeslot == NULL)
goto found_unused; // -> 15
so->used++;
freeslot->key = key; // -> 16
freeslot->hash = hash; // -> 17
return 0;

found_unused:
so->fill++; // -> 18
so->used++; // -> 19
entry->key = key; // -> 20
entry->hash = hash; // -> 21
if ((size_t)so->fill*5 < mask*3)
return 0;

return set_table_resize(so, so->used>50000 ? so->used*2 : so->used*4); // -> 22
...
}
```

Code insights are as

Search for the index at the position at the hash masked by the size of the set.

Inspect the element at the hashed value.

If the element at the hashed value is empty, try inserting the element at the particular index.

Continue the search of the element until either the element is found or an empty slot can be found to insert the value to the set.

If the hash being inserted is equal to the hash of the element being inserted, check if the same element is at the current location.

Check for equality of integer types, and if equal, detect the element to be found.

Check for equality of Unicode strings, and if equal, detect the element to be found.

Check hashable objects such as tuples for equality and detect the element to be found if present.

Same as

If the element is not present at the same location, search for a slot closer to the hashed index.

Increment the index to the next elements in the array.

Search for the element in the new index very similar to the preceding comments.

If a free slot is found, break the loop to insert the element at the found index.

Same as

Same as

Assign the key to be the element at the current index.

Assign the hash of the element at the index but do not increment the size as we are assigning at an element where the value was preassigned but removed.

Increase the elements filled in the set as we are inserting at a position not previously assigned.

Increase the elements being used in the set as we are inserting at a position not previously assigned.

Same as

Same as

Resize the elements if the number of elements is greater than the size of the set.

Iterating a set

Iterating a set involves going through all the elements in the set mostly without order as the elements are inserted into the array based on the hash value.

Code sample is follows:

```
a = {1, 2, 3}

for ele in a:
print(ele)
...
```

Opcode

```
5           10 LOAD_NAME                0 (a)
12 GET_ITER                             //   -> 1
>>     14 FOR_ITER                  12 (0 28)   // -> 2
16 STORE_NAME                1 (ele)
...
```

Code insights are as

Fetch the iterator of the set.

Loop through the iterator using the **FOR_ITER** opcode.

The following code block demonstrates the structure and functions of the **set** iterator:

Objects/setobject.c (line no 805)

```
typedef struct {
PyObject_HEAD
PySetObject *si_set; /* Set to NULL when iterator is exhausted */
// -> 1
Py_ssize_t si_used; // -> 2
Py_ssize_t si_pos; // -> 3

Py_ssize_t len; // -> 4
} setiterobject;

static PyObject* set_iter(PySetObject *so)
{
setiterobject *si = PyObject_GC_New(setiterobject, &PySetIter_Type);
// -> 5
..
si->si_set = so; // -> 6
si->si_used = so->used; // -> 7
si->si_pos = 0; // -> 8
si->len = so->used; // -> 9
...
}
```

Code insights are as

The **set** object is being iterated.

The number of elements in the set.

The starting position to start the iteration in the set.

The length of the set is not used in the iteration.

Obtain the memory for the iteration.

The set to insert the element into.

Assign the number of elements in the set to be equal to the **si_used** variable in the **set** object.

Assign the starting position to be equal to

The length of the set is equal to the number of used elements in the set.

The following code block demonstrates the iteration of elements:

Objects/setobject.c (line no 867)

```
static PyObject *setiter_iternext(setiterobject *si)

{
...
```

```
PySetObject *so = si->si_set; // -> 1
...

i = si->si_pos; // - > 2
...
while (i <= mask && (entry[i].key == NULL || entry[i].key ==
dummy)) // -> 3
i++;
si->si_pos = i+1; // -> 4
...
si->len--;
key = entry[i].key; // -> 5
...
}
```

Code insights are as

The **set** object is being iterated.

The current element to start iteration from.

Skip the elements with either dummy or **NULL** entries in the array.

The next position to start iteration from in the next iteration loop.

The key is to be returned to the current position.

Finding an element in a set

A set is a hash-based data structure that performs better than a list when searching for an element within it.

Code sample is as follows:

```
a = {1, 2, 3}

if 3 in a:
print("Element found")
```

```
...
14 COMPARE_OP               6 (in) // -> 1
16 POP_JUMP_IF_FALSE        26
...
```

The code insight is as

The **COMPARE_OP** is used to check if the element exists in the set.

The following code block explains the implementation of the comparison operator:

Objects/setobject.c (line no 56)

```
static setentry* set_lookkey(PySetObject *so, PyObject *key,
Py_hash_t hash)
{
...


...
}
```

The code block implementation is exactly similar to inserting an element into a set except where the element is checked to be present in the set. It is suggested to go through the implementation from the source code to identify the similarity.

Union and intersection of sets

Sets have the unique capability to quickly identify the elements common to both sets and either construct a union with elements in both sets and eliminate the duplicates or intersection containing only the elements common to both sets. This section covers the implementation of these operations.

The following code block demonstrates the implementation of the **union** in set objects:

Objects/setobject.c (line no 1177)

```
static PyObject* set_union(PySetObject *so, PyObject *args)
{
...
result = (PySetObject *)set_copy(so, NULL); // -> 1
if (result == NULL)
return NULL;

for (i=0 ; i; i++) {
other = PyTuple_GET_ITEM(args, i);
if ((PyObject *)so == other) // -> 2
continue;
if (set_update_internal(result, other)) {// -> 3
...
}
```

```
}
return (PyObject *)result;
}
```

Code insights are as

Copy the first set into the resulting set to ensure all elements of the first set are present.

Check if the element already exists in the resulting set.

Update the element if not present.

The following code block demonstrates the implementation of intersection in **set** objects:

Objects/setobject.c (line no 1225)

```
static PyObject* set_intersection(PySetObject *so, PyObject *other)
{

...
result = (PySetObject *)make_new_set_basetype(Py_TYPE(so),
NULL); // -> 1
...

if (PyAnySet_Check(other)) {// -> 2
...
```

```
while (set_next((PySetObject *)other, &pos, &entry)) {// -> 3
key = entry->key;
hash = entry->hash;
rv = set_contains_entry(so, key, hash); // -> 4
if (rv < 0) {
Py_DECREF(result);
return NULL;
}
if (rv) {
if (set_add_entry(result, key, hash)) {// -> 5
Py_DECREF(result);
return NULL;
}
}
}

return (PyObject *)result;
}

it = PyObject_GetIter(other);
...
while ((key = PyIter_Next(it)) != NULL) {// -> 6
hash = PyObject_Hash(key); // -> 7
...
rv = set_contains_entry(so, key, hash); // -> 7
...
set_add_entry(result, key, hash)) // -> 8
...
```

Code insights are as

Create an empty set with the intersection of values.

The intersection can be performed on iterable Python types. If the second object is a set, we use set operations for checking if an element exists in the two sets.

Fetch the element from *set*

Check if the element exists in *set*

Add the element which exists in both the sets into the resultant set.

Fetch the next object if it is not a list.

Fetch the hash of the element in the second object.

Check if the element exists in the second set and insert if it exists in both the iterable types.

Dictionaries

Dictionaries are key-value pairs that are saved based on the hash value of the key into an array very similar to sets. The section displays the similarities between a dictionary and a set-in structure and storage. Dictionaries differ in the way that the value stored in a dictionary can be any valid Python object while the key has to be a hashable type.

Structure of a dictionary

Although semantically similar to a set, the structure of a Python dictionary exhibits differences from the structure of a set:

Objects/dict-common.h (line no 22)
struct _dictkeysobject {
Py_ssize_t dk_refcnt;

/* Size of the hash table (dk_indices). It must be a power of 2. */
Py_ssize_t dk_size; // -> 1

...
dict_lookup_func dk_lookup; // -> 2

/* Number of usable entries in dk_entries. */
Py_ssize_t dk_usable; // -> 3

/* Number of used entries in dk_entries. */
Py_ssize_t dk_nentries; // -> 4

...
char dk_indices[]; /* char is required to avoid strict aliasing. */
// -> 5

```
/* "PyDictKeyEntry dk_entries[dk_usable];" array follows:
see the DK_ENTRIES() macro */
};
```

Objects/dictobject.h (line no 14)

```
typedef struct {
PyObject_HEAD

/* Number of items in the dictionary */
Py_ssize_t ma_used; // -> 6

...
PyDictKeysObject *ma_keys; // -> 7

...

PyObject **ma_values; // -> 8
} PyDictObject;
```

Code insights are as

The size/storage capacity of the dictionary.

The **lookup** function is to search for keys in the table.

The number of usable entries in the hash table that are not a dummy.

The number of used entries in the hash table.

The key of the object.

The number of items stored in the dictionary.

Array to store the keys in the dictionary.

Array to store the values in the dictionary.

Creating and inserting to dictionaries

Creating a dictionary involves allocating memory to the hash table to store the keys and values. The key must be any hashable value, whereas the value can be any valid Python object:

```
a = {1: 1, 2: 2, 3: 3}
2           0 LOAD_CONST               0 (1)
            2 LOAD_CONST               1 (2)
            4 LOAD_CONST               2 (3)
            6 LOAD_CONST               3 ((1, 2, 3)) // -> 1
            8 BUILD_CONST_KEY_MAP      3 // -> 2
           10 STORE_NAME               0 (a)
           12 LOAD_CONST               4 (None)
           14 RETURN_VALUE
```

Code insight is as

The **BUILD_CONST_KEY_MAP** is used to build the dictionary.

Python/ceval.c (line no 2867)

```
case TARGET(BUILD_CONST_KEY_MAP): {

Py_ssize_t i;
PyObject *map;
```

```
PyObject *keys = TOP(); // -> 1
....
map = _PyDict_NewPresized((Py_ssize_t)oparg); // -> 2
...
for (i = oparg; i > 0; i--) {
int err;
PyObject *key = PyTuple_GET_ITEM(keys, oparg - i); // -> 3
PyObject *value = PEEK(i + 1); // -> 4

err = PyDict_SetItem(map, key, value); // -> 5
...
}
...
}
```

Code insights are as

Fetch the keys to be inserted into the dictionary.

The creation of the new dictionary will be explained in the following code.

Fetch the key from the tuple created.

Fetch the value from the function value stack.

Insert the value into the dictionary.

Objects/dictobject.c (line no 1314)

```
PyObject* _PyDict_NewPresized(Py_ssize_t minused)
{
const Py_ssize_t max_presize = 128 * 1024;
Py_ssize_t newsize;
PyDictKeysObject *new_keys;
...
assert(IS_POWER_OF_2(newsize)); // -> 1
...
new_keys = new_keys_object(newsize); // -> 2
if (new_keys == NULL)
return NULL;
return new_dict(new_keys, NULL); // -> 3
}
```

Objects/dictobject.c (Line no 530)

```
static PyDictKeysObject *new_keys_object(Py_ssize_t size)
{

PyDictKeysObject *dk;
Py_ssize_t es, usable;

assert(size >= PyDict_MINSIZE);
assert(IS_POWER_OF_2(size));

usable = USABLE_FRACTION(size);
...
_Py_INC_REFTOTAL;
dk->dk_refcnt = 1;
dk->dk_size = size; // -> 1
dk->dk_usable = usable; // -> 2
```

```
dk->dk_lookup = lookdict_unicode_nodummy; // -> 3
dk->dk_nentries = 0;
memset(&dk->dk_indices[0], 0xff, es * size);
memset(DK_ENTRIES(dk), 0, sizeof(PyDictKeyEntry) * usable); // ->
4
return dk;
}

static PyObject* new_dict(PyDictKeysObject *keys, PyObject
**values)
{
PyDictObject *mp;
assert(keys != NULL);
...
mp = PyObject_GC_New(PyDictObject, &PyDict_Type); // -> 5
...
mp->ma_keys = keys; // -> 6
mp->ma_values = values; // -> 7
mp->ma_used = 0;


...
return (PyObject *)mp;
}
```

Code insights are as

Initialize the size of the hash table to the size of the created hash table in this case

Assert the power of the dictionary to be a power of **2** for easier memory management.

Initialize the **lookup** function to be equal to the function **lookdict_unicode_nodummy** to be initialized in the next section.

Initialize all the values to be equal to

Create a new object of the type

Assign the **keys** object to the keys obtained.

Assign the **values** object to the values provided to the program.

Objects/dictobject.c (line no 1522)

```
int PyDict_SetItem(PyObject *op, PyObject *key, PyObject *value)
{
PyDictObject *mp;
Py_hash_t hash;
if (!PyDict_Check(op)) {
PyErr_BadInternalCall();
return -1;
}
...
if (!PyUnicode_CheckExact(key) ||
(hash = ((PyASCIIObject *) key)->hash) == -1)

{
hash = PyObject_Hash(key); // -> 1
if (hash == -1)
return -1;
```

```
}
...
/* insertdict() handles any resizing that might be necessary */
return insertdict(mp, key, hash, value); // -> 2
}
```

Objects/dictobject.c (line no 1027)

```
static int insertdict(PyDictObject *mp, PyObject *key, Py_hash_t
hash, PyObject *value)
{
PyObject *old_value;
PyDictKeyEntry *ep;


...
if (mp->ma_values != NULL && !PyUnicode_CheckExact(key)) {
if (insertion_resize(mp) < 0)
goto Fail;
}


Py_ssize_t ix = mp->ma_keys->dk_lookup(mp, key, hash,
&old_value); // -> 3
...
if (ix == DKIX_EMPTY) {// -> 4
...
Py_ssize_t hashpos = find_empty_slot(mp->ma_keys, hash); // -> 5


...
dictkeys_set_index(mp->ma_keys, hashpos, mp->ma_keys-
>dk_nentries); // -> 6
ep->me_key = key; // -> 7
```

```
ep->me_hash = hash; // -> 8
if (mp->ma_values) {
assert (mp->ma_values[mp->ma_keys->dk_nentries] == NULL);
mp->ma_values[mp->ma_keys->dk_nentries] = value; // -> 10
}
else {
ep->me_value = value;
}
mp->ma_used++;

mp->ma_keys->dk_nentries++; // -> 9
...
}
...
}
```

Code insights are as

Fetch the hash of the object to be inserted into the table.

Use the **insertdict** function to insert the value into the dictionary.

lookup function to fetch the key of the value to be inserted into the dictionary.

Check if the slot is empty.

Find an empty slot to insert the value to.

Set the value at the **hashpos** to the value at the index.

Set the key and value.

Same as

From Python 3 onwards, the dictionaries are ordered by nature; hence the number of entries is incremented after inserting to preserve insertion order.

Add the value into the **values** array to preserve the insertion order.

Iterating dictionaries

Iterating dictionaries previous to Python 3 were not ordered by nature, and hence the traversal order would be different from the insertion order. From Python 3, the insertion order is maintained during traversal.

Code sample is as follows:

```
a = {1: 1, 2: 2, 3: 3, 4: 4}

for i in a.items():
print(i)
3           14 LOAD_NAME                0 (a)
16 LOAD_METHOD              1 (items)
18 CALL_METHOD              0
20 GET_ITER                 // -> 1
>> 22 FOR_ITER              12 (to 36) // -> 2
24 STORE_NAME               2 (i)
...
```

Code insights are as

Obtain the iterator for the dictionary explained in the following section.

Iterate through the dictionary.

Objects/dictiterobject.c (line no 3446)

```
typedef struct {
PyObject_HEAD
PyDictObject *di_dict; /* Set to NULL when iterator is exhausted
*/ // -> 1
Py_ssize_t di_used;
Py_ssize_t di_pos;   // -> 2
PyObject* di_result; /* reusable result tuple for iteritems */ // ->
3

Py_ssize_t len;
} dictiterobject;
```

Objects/dictiterobject.c (line no 3456)

```
static PyObject* dictiter_new(PyDictObject *dict, PyTypeObject
*itertype)
{
dictiterobject *di;
di = PyObject_GC_New(dictiterobject, itertype); // -> 4
if (di == NULL) {
return NULL;
}
Py_INCREF(dict);
di->di_dict = dict; // -> 5
di->di_used = dict->ma_used; // -> 6

...
_PyObject_GC_TRACK(di);
```

```
return (PyObject *)di;
}

static PyObject* dictiter_iternextkey(dictiterobject *di)
{
PyObject *key;
Py_ssize_t i;
PyDictKeysObject *k;
PyDictObject *d = di->di_dict;
...

i = di->di_pos; // -> 7
k = d->ma_keys; // -> 8
assert(i >= 0);



Py_ssize_t n = k->dk_nentries;
PyDictKeyEntry *entry_ptr = &DK_ENTRIES(k)[i]; // -> 9

while (i < n && entry_ptr->me_value == NULL) {// -> 10
entry_ptr++;
i++;
}
...

di->di_pos = i+1; // -> 11
...
return key; // -> 12
...
}
```

Code insights are as

The structure of the **dictiterobject** contains access to the dictionary to iterate.

The position to start the iteration from.

The result tuple for iterating the items in the dictionary.

Allocate memory to a new instance of the iterator type.

The dictionary to iterate.

The number of items to iterate.

The position to fetch the current element from.

The keys in the dictionary.

Fetch the entry at the current pointer.

Iterate the dictionary until a element is found.

Move the position to the next element to continue the iteration.

Return the key to fetch the item from.

Conclusion

This chapter covered the most commonly used hashable data structures in Python, which are the set and dictionaries. Sets are data structures, which are used primarily for the purpose of searching an element in a large collection of items. The structure of the set object was covered and how it contains a reference to a hash table to store the elements in the object. The main use case of a set object is to search for an element in a list of elements. Sets can also be subjected to union, intersection, and other operations.

A dictionary object can be used to store data as a key, value pairs for *O(1) insertion and* Insertion, search, and iteration of elements work similarly to set objects as they contain a similar implementation.

The upcoming chapter will discuss the structure of the function and generator objects and how they are created at runtime. The function object contains a link to the code object, that is, linked to the data provided to the function to create a stack object and execute the function call. Generators are functions whose execution can be stopped in-between, which is implemented as the stack frame position is stored at the point of yielding from the generator.

CHAPTER 5

Functions and Generators

In the previous chapter, we covered the structure and design of sets and dictionaries, including hashing and the implementation of functions, which handle insertion, indexing, and removal of elements from these data structures.

This chapter covers functions that form the heart of reusable code in programming languages and contain a definition and returns values based on the accepted arguments. Functions must return the same value for a particular set of arguments, and once declared, they can be called from any stage of a program by passing arguments.

Structure

In this chapter, we will cover the following topics:

Creation of the **PyFunctionObject**

The **LOAD_CONST** opcode

The **MAKE_FUNCTION** opcode

Function call

Structure of a function frame

The **CALL_FUNCTION** opcode

Generators

Creating a generator

Creating an instance of a generator object

Structure of the generator object

Execution of a generator

Execution of the generator code

Objective

After studying this chapter, you will be able to visualize the structure of a function object in memory. You will be able to understand the lifecycle of a function call, the creation of a stack frame, and the execution of the interpreter using the created stack frame. You will also learn the structure of a generator object in memory and the internal workings of the yield operation.

Refreshing your knowledge on Python generators if unfamiliar will help to understand the functioning better.

Creation of the PyFunctionObject

The **PyFunctionObject** contains all the unchanging attributes of a function like **code** and so on. A Python stack frame is created every time the function is called, whereas a **PyFunctionObject** is created once:

Create a file function.py

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

Opcode 7.1

```
$ python -m dis function.py
1           0 LOAD_CONST               0 (object sum at
0x10d43e870, file "func.py", line 1>)
2 LOAD_CONST                1 ('sum')
4 MAKE_FUNCTION             0
6 STORE_NAME                0 (sum)
```

The **MAKE_FUNCTION** opcode creates the reusable **PyFunctionObject,** the structure of which is explained as follows:

Include/funcobject.h (Line no 21)

```
typedef struct {
```

```
PyObject_HEAD
PyObject *func_code;            /* A code object, the __code__
attribute */ // -> 1
PyObject *func_globals;         /* A dictionary (other mappings won't
do) */ // -> 2
PyObject *func_defaults;        /* NULL or a tuple */   // -> 3
PyObject *func_kwdefaults;      /* NULL or a dict */ // -> 4
PyObject *func_closure;         /* NULL or a tuple of cell objects */
// -> 5

PyObject *func_doc;             /* The __doc__ attribute, can be
anything */ // -> 6
PyObject *func_name;            /* The __name__ attribute, a string
object */ // -> 7
PyObject *func_dict;            /* The __dict__ attribute, a dict or
NULL */ -> 8
PyObject *func_weakreflist; /* List of weak references */ -> 9

PyObject *func_annotations; /* Annotations, a dict or NULL */ ->
10
vectorcallfunc vectorcall; // -> 11


/* Invariant:
*       func_closure contains the bindings for func_code-
>co_freevars, so
*       PyTuple_Size(func_closure) ==
PyCode_GetNumFree(func_code)
*       (func_closure may be NULL if
PyCode_GetNumFree(func_code) == 0).
*/
} PyFunctionObject;
```

Code insights are as

The **func_code** is a pointer to the **code** object of the function, which contains the byte code to be executed by the interpreter.

The **globals** pointer contains the dictionary to the global variables, which can be used by the function.

func_defaults pointer contains the pointer to a tuple of values, which are the default values of arguments to the function.

The **func_kwdefaults** pointer contains the pointer to a dictionary of values, which are the keyword arguments to the function.

A tuple of cell objects contains the values used by the enclosing functions.

The documentation of any of the function.

The function name.

All functions contain all the arguments passed as the **__dict__** of the function.

List of weak references to the function.

List of annotations of arguments and return values of the function.

The **vectorcall** function is used to create a stack of the executing function and execute the function call.

The LOAD_CONST opcode

The Python interpreter, being stack-based, uses the **LOAD_CONST** opcode to add a value to the head of the data stack pointer. The data loaded into the stack are used by later opcodes, such as the addition of two numbers that loads two numbers to the stack and calls the **BINARY_ADD** opcode to use these values. Opcodes will be covered in detail in *Chapter 7 on Interpreter and*

The following code block explains the implementation of the **LOAD_CONST** opcode and the **BASIC_PUSH** macro:

Python/ceval.c (Line no 1346)

```
case TARGET(LOAD_CONST): {
PREDICTED(LOAD_CONST);
PyObject *value = GETITEM(consts, oparg);
Py_INCREF(value);
PUSH(value);
FAST_DISPATCH();
}
```

Python/ceval.c (Line no 988)

```
#define PUSH(v)                BASIC_PUSH(v)
#define BASIC_PUSH(v)          (*stack_pointer++ = (v))
```

The preceding code block demonstrates that the object is added to the head of the *Opcode 7.1* pushes the code object and the name of the function into the stack.

The MAKE_FUNCTION opcode

The opcode **MAKE_FUNCTION** creates the **PyFunctionObject** using the constructor function

Python/ceval.c (Line no 3571)

```
case TARGET(MAKE_FUNCTION): {
...
codeobj = POP();
qualname = POP();
PyFunctionObject *func = (PyFunctionObject *)
PyFunction_NewWithQualName(codeobj, f->f_globals, qualname);
if (func == NULL) {
goto error;
}
...

PUSH((PyObject *) func);
DISPATCH();
}
```

Code insights are as

The **MAKE_FUNCTION** opcode calls the **PyFunction_NewWithQualName** that takes three parameters:

The code object that contains the opcode to be executed for the function.

The global variables.

The name of the function to be created.

Objects/funcobject.c (Line no 13)

```
PyObject* PyFunction_NewWithQualName(PyObject *code, PyObject
*globals, PyObject *qualname)
{

PyFunctionObject *op;
....

op = PyObject_GC_New(PyFunctionObject, &PyFunction_Type); // ->
1
if (op == NULL)
return NULL;

op->func_weakreflist = NULL;
Py_INCREF(code);
op->func_code = code; // -> 2
Py_INCREF(globals);
op->func_globals = globals; // -> 3
op->func_name = ((PyCodeObject *)code)->co_name; // -> 4
Py_INCREF(op->func_name);
....
```

```
op->func_closure = NULL;
op->vectorcall = _PyFunction_Vectorcall; // -> 5

consts = ((PyCodeObject *)code)->co_consts; // -> 6
...

/* __module__: If module name is in globals, use it.
Otherwise, use None. */
module = PyDict_GetItemWithError(globals, __name__);

if (module) {
Py_INCREF(module);
op->func_module = module; // -> 7
}

....
if (qualname)

op->func_qualname = qualname;
else
op->func_qualname = op->func_name;
Py_INCREF(op->func_qualname);

_PyObject_GC_TRACK(op);
return (PyObject *)op;
}
```

Code insights are as

The **PyObject_GC_New(PyFunctionObject, &PyFunction_Type)** call allocates memory to the Python function object on the heap.

Assign the code object to the **func_code** reference in the

Assign the **globals** for usage to the **globals** reference in the

Assign the name to the **func_name** reference. The name of the function is stored in the code object during the compilation phase.

The **Vectorcall** function creates the stack frame and calls the interpreter to schedule the frame for execution. This process will be covered in detail in the next section.

Assign the **co_consts** to the reference in the

Assigns the **func_module** to the reference in the **PyFunctionObject** and derives the same from the **globals** object.

Function call

The once created, can be called multiple times, passing different parameters. Each time the function is called, it creates a new instance of a function frame and is added to the head of the frame stack for the currently executing thread. Once the execution is completed, the frame is deallocated, and the previous frame resumes execution.

Structure of a function frame

The following code block explains the structure of a Python frame:

Objects/frameobject.c (Line no 16)

```
typedef struct _frame {
PyObject_VAR_HEAD
struct _frame *f_back;     /* previous frame, or NULL */ -> 1
PyCodeObject *f_code;      /* code segment */     / -> 2
PyObject *f_builtins;      /* builtin symbol table (PyDictObject) */
PyObject *f_globals;       /* global symbol table (PyDictObject) */
/ -> 3
...
/* Borrowed reference to a generator, or NULL */
PyObject *f_gen;

int f_lasti;                    /* Last instruction if called */ / -> 4
/* Call PyFrame_GetLineNumber() instead of reading this field
directly.  As of 2.3 f_lineno is only valid when tracing is
active (i.e. when f_trace is set).  At other times we use
PyCode_Addr2Line to calculate the line from the current
bytecode index. */
int f_lineno;                   /* Current line number */ -> 5
int f_iblock;                   /* index in f_blockstack */
char f_executing;               /* whether the frame is still executing
*/
```

```
PyTryBlock f_blockstack[CO_MAXBLOCKS]; /* for try and loop
blocks */
PyObject *f_localsplus[1];  /* locals+stack, dynamically sized */

} PyFrameObject;
```

Code insights are as

The preceding code describes the following important items, covering the rest is beyond the scope of this book:

Pointer to the previous frame in the Python stack frame. Post execution of the current frame, the thread moves to execute the frame pointed by

Pointer to the code object of the executing Python function.

Pointer to the global symbol table.

Pointer to the last instruction executed in the **code** object.

Line number currently being executed.

CALL_FUNCTION opcode

Sample Python function that calls the function sum with arguments

Create a file function_call.py

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

sum_nos = sum(10, 20)
```

Opcode 7.2

```
$ python -m dis function_call.py
3           0 LOAD_CONST               0 (object sum at
0x10135687O, file "func_call.py", line 3>)
2 LOAD_CONST               1 ('sum')
4 MAKE_FUNCTION            0
6 STORE_NAME               0 (sum)
6           8 LOAD_NAME                0 (sum)
10 LOAD_CONST              2 (10)
12 LOAD_CONST              3 (20)
14 CALL_FUNCTION           2
16 STORE_NAME              1 (sum_nos)
18 LOAD_CONST              4 (None)
20 RETURN_VALUE
```

Disassembly of object sum at 0x101356870, file "func_call.py", line 3>:

4 0 LOAD_FAST 0 (a)

2 LOAD_FAST 1 (b)

4 BINARY_ADD

Calling a function invokes the **CALL_FUNCTION** opcode. Implementation of the **CALL_FUNCTION** opcode:

Python/ceval.c (Line no 3496)

```
case TARGET(CALL_FUNCTION): {
PREDICTED(CALL_FUNCTION);
PyObject **sp, *res;
sp = stack_pointer;
res = call_function(tstate, &sp, oparg, NULL);
stack_pointer = sp;
PUSH(res);
if (res == NULL) {
goto error;
}
DISPATCH();
}
```

The tracing of the **call_function** runs into several stages, but it internally calls the **_PyFunction_Vectorcall** function, which handles stack frame creation and passing into the interpreter:

```
PyObject * _PyFunction_Vectorcall(PyObject *func,
```

```
PyObject* const* stack,
size_t nargsf,
PyObject *kwnames)
{
PyCodeObject *co = (PyCodeObject *)PyFunction_GET_CODE(func);
PyObject *globals = PyFunction_GET_GLOBALS(func);
PyObject *argdefs = PyFunction_GET_DEFAULTS(func);
...

kwdefs = PyFunction_GET_KW_DEFAULTS(func);
closure = PyFunction_GET_CLOSURE(func);
name = ((PyFunctionObject *)func) -> func_name;
qualname = ((PyFunctionObject *)func) -> func_qualname;



...

return _PyEval_EvalCodeWithName((PyObject*)co, globals, (PyObject
*)NULL, stack, nargs ...); // -> 1
}

Python/ceval.c (line no 4045)

PyObject* _PyEval_EvalCodeWithName(PyObject *_co, PyObject
*globals, PyObject ...)
{
PyCodeObject* co = (PyCodeObject*)_co;
PyFrameObject *f;
...
```

```
/* Create the frame */
f = _PyFrame_New_NoTrack(tstate, co, globals, locals); // -> 2
if (f == NULL) {
return NULL;
}
fastlocals = f->f_localsplus;
freevars = f->f_localsplus + co->co_nlocals;


...
...


retval = PyEval_EvalFrameEx(f,0); // -> 3
fail: /* Jump here from prelude on failure */
....
else {
++tstate->recursion_depth;
Py_DECREF(f);
--tstate->recursion_depth;
}
return retval;


}
```

Code insights are as

Call the interpreter function to indicate the execution of the function call.

Creation of the stack frame for execution of the function call.

Calls the main interpreter code interprets the opcode of the function one after the other. The execution of the opcodes by the interpreter is covered in the *Chapter 7 on Interpreter and*

Generators

Generators are special functions whose execution can be stopped in-between using the **yield** keyword and can be re-invoked to continue execution. The concept used in Python to implement generators is to halt the function frame, which contains the current line number and opcode of execution. When the next function is called to continue execution, this saved state resumes operations from the point, it left off.

Creating a generator

Sample Python code to create a generator that yields values from **0** to

```
def create_gen():
for i in range(0, 10):
yield i
2           0 LOAD_CONST               0 (object create_gen
at 0x10f70d870, file "create_generator.py", line 2>)
2 LOAD_CONST               1 ('create_gen')
4 MAKE_FUNCTION            0
6 STORE_NAME               0 (create_gen)
8 LOAD_CONST               2 (None)
10 RETURN_VALUE
```

Disassembly of object create_gen at 0x10f70d870, file "create_generator.py", line 2>:

```
3           0 LOAD_GLOBAL              0 (range)
2 LOAD_CONST               1 (0)
4 LOAD_CONST               2 (10)
6 CALL_FUNCTION            2
8 GET_ITER
>>    10 FOR_ITER                10 (to 22)
12 STORE_FAST               0 (i)
```

The **MAKE_FUNCTION** opcode uses the compiler flag to denote the type being a generator. Although normal function calls invoke the interpreter when called, the generator functions return an instance of the More about this is covered in the upcoming sections on generator instances and invoking them.

The following code object indicates the compiler adding the **CO_GENERATOR** flag:

Python/compile.c (line no 5810)

```
PySTEntryObject *ste = c->u->u_ste;
int flags = 0;
if (ste->ste_type == FunctionBlock) {
flags |= CO_NEWLOCALS | CO_OPTIMIZED;
if (ste->ste_nested)
flags |= CO_NESTED;
if (ste->ste_generator && !ste->ste_coroutine)
flags |= CO_GENERATOR;
if (!ste->ste_generator && ste->ste_coroutine)
flags |= CO_COROUTINE;
...
```

Covering how the compiler creates the **Abstract Syntax Tree** is beyond the scope of this book. However, the flag in the code object is used while creating the stack frame for the generator.

Creating an instance of a generator object

Generators, although similar to functions, require an instance to handle the execution of the opcodes within them. Multiple instances of the generator instances can be created with each at different execution stages.

```
def create_gen():
for i in range(0, 10):
yield i
```

gen_obj = create_gen()

```
next(gen_obj)
```

```
2           0 LOAD_CONST               0 (object create_gen
at 0x1096a1870, file "gen_call.py", line 2>)
2 LOAD_CONST                1 ('create_gen')
4 MAKE_FUNCTION             0
6 STORE_NAME                0 (create_gen)
6           8 LOAD_NAME                0 (create_gen)
10 CALL_FUNCTION            0    //    -> 1
12 STORE_NAME               1 (gen_obj)

7          14 LOAD_NAME                2 (next)
16 LOAD_NAME                1 (gen_obj)
18 CALL_FUNCTION            1    //    -> 2
```

```
20  POP_TOP
22  LOAD_CONST                2  (None)
24  RETURN_VALUE
```

Disassembly of object create_gen at 0x1096a1870, file "gen_call.py", line 2>:

```
3           0 LOAD_GLOBAL              0 (range)
2 LOAD_CONST               1 (0)     // -> 3

4 LOAD_CONST               2 (10)
6 CALL_FUNCTION            2
8 GET_ITER
>>    10 FOR_ITER                10 (to 22)
12 STORE_FAST              0 (i)

4           14 LOAD_FAST                0 (i)
16 YIELD_VALUE
18 POP_TOP
20 JUMP_ABSOLUTE           10
>>    22 LOAD_CONST               0 (None)
24 RETURN_VALUE
```

This opcode is used in all the following explanations referred to as the *Gen Opcode*

A generator object can be created by calling the function. The following code explains the difference between a routine function call and a generator:

Python/ceval.c (line no 4044)

```
PyObject *
_PyEval_EvalCodeWithName(PyObject *_co, PyObject *globals,
PyObject *locals,
PyObject *const *args, Py_ssize_t argcount,
Py_ssize_t kwcount, ....)
{
...

/* Create the frame */
f = _PyFrame_New_NoTrack(tstate, co, globals, locals); // -> 1
if (f == NULL) {
return NULL;
}
```

Python/ceval.c (line no 4272)

```
...
/* Handle generator/coroutine/asynchronous generator */
if (co->co_flags & (CO_GENERATOR | CO_COROUTINE |
CO_ASYNC_GENERATOR)) {// -> 2
PyObject *gen;
int is_coro = co->co_flags & CO_COROUTINE;
...
gen = PyGen_NewWithQualName(f, name, qualname); // -> 3
}
...
_PyObject_GC_TRACK(f);
return gen; // -> 4
}
```

retval = PyEval_EvalFrameEx(f,o); // -> 4
fail: /* Jump here from prelude on failure */
...

Code insights are as

Creation of a new stack frame is created for a regular function call.

The **CO_GENERATOR** flag, as discussed in the previous section, was added at compile time.

A new instance of a Python generator object is created with the newly created stack frame.

The code for the frame is not executed by the interpreter as in the previous section, but the newly created generator object has been returned.

Structure of the generator object

The section covers the structure of the generator object and its attributes and how an executing frame is stopped in between execution when yielded:

```
#define _PyGenObject_HEAD(prefix)                                   \
PyObject_HEAD                                                       \
/* Note: gi_frame can be NULL if the generator is "finished" */     \
struct _frame *prefix##_frame;                                      \ // -> 1
/* True if generator is being executed. */                          \
char prefix##_running;                                              \
/* The code object backing the generator */                        \
PyObject *prefix##_code;                                            \ // -> 2
/* List of weak reference. */                                       \
PyObject *prefix##_weakreflist;                                     \
/* Name of the generator. */                                        \ // -> 3
PyObject *prefix##_name;                                            \
/* Qualified name of the generator. */                              \
PyObject *prefix##_qualname;                                        \
_PyErr_StackItem prefix##_exc_state;
typedef struct {
/* The gi_ prefix is intended to remind of generator-iterator. */   // -> 4
_PyGenObject_HEAD(gi)
} PyGenObject;
```

Code insight are as

The stack frame contains the function code that will be executed from the point it left off.

The code object will be executed, although it is primarily stored only for reference.

The name of the generator will be the name of the assigned variable.

The expansion of the macro into the actual generator structure of the The macro is also used for **AsyncGenObject** and **AsyncIterGenObject,** which internally use generators to create coroutines that await until the completion of execution of the awaited function. Asynchronous programming in Python is covered in *Chapter 9 on Async*

Execution of a generator

The created generator instance is inert until the **next** function is called on it that starts executing the stack frame until yielding a value from the function:

From	**Gen Opcode**	**Sample**
7	14 LOAD_NAME	2 (next)
16 LOAD_NAME	1 (gen_obj)	
18 CALL_FUNCTION	1 // -> 2	

The logic to enter and yield from the generator frame is handled by the **next** function; the declaration and implementation of which is contained in the following code:

Python/bltinmodule.c (Line no 2739)

{"next", (PyCFunction)(void(*)(void))builtin_next, METH_FASTCALL, next_doc}

Python/bltinmodule.c (Line no 1373)

```
static PyObject* builtin_next(PyObject *self, PyObject *const *args,
Py_ssize_t nargs)
{
PyObject *it, *res;
```

....

res = (*it->ob_type->tp_iternext)(it); // -> 1

```
if (res != NULL) {
return res;
}
....
PyErr_SetNone(PyExc_StopIteration); // -> 2
return NULL;
....
}

}
```

Code insights are as

The function calls the **tp_iternext** function with the generator as the argument. The implementation of the same will be discussed in the next code section.

The **StopIterationException** is raised indicates the completion of execution of the generator.

Objects/genobject.c (Line no 153)

```
static PyObject* gen_send_ex(PyGenObject *gen, PyObject *arg, int
exc, int closing)
{
....
```

```
f->f_back = tstate->frame; // -> 1


gen->gi_running = 1; // -> 2
gen->gi_exc_state.previous_item = tstate->exc_info; // -> 3
tstate->exc_info = &gen->gi_exc_state; // -> 4
result = PyEval_EvalFrameEx(f, exc); // -> 5
tstate->exc_info = gen->gi_exc_state.previous_item; // -> 6
gen->gi_exc_state.previous_item = NULL;
gen->gi_running = 0; // -> 7


/* Don't keep the reference to f_back any longer than necessary.
It
* may keep a chain of frames alive or it could create a reference
* cycle. */
assert(f->f_back == tstate->frame);
Py_CLEAR(f->f_back);


....
return result;
}
```

Code insights are as

Mark the current frame being executed on the thread as the frame behind the generator before executing it.

Mark the generator currently running before starting/continuing the execution.

Copy the current exception in the stack to the **previous_item** in the generator exception stack.

Copy the current exception stack of the generator frame to the currently executing thread state.

Start interpreting the generator stack frame until the function yields a value or raises the **StopIterationException.**

Replace back the thread state with the previous thread state information before executing the generator.

Set the generator flag to not run at the current state.

Execution of the generator code

As explained previously, the execution of the generator function is started by the interpreter, and the function code is executed until the point of raising the

```
From          Gen  Opcode                Sample
3             0    LOAD_GLOBAL           0  (range)
              2    LOAD_CONST            1  (0)
              4    LOAD_CONST            2  (10)
              6    CALL_FUNCTION         2
              8    GET_ITER
        >>    10   FOR_ITER              10 (to 22)
              12   STORE_FAST            0  (i)
4             14   LOAD_FAST             0  (i)
              16   YIELD_VALUE
              18   POP_TOP
              20   JUMP_ABSOLUTE         10
        >>    22   LOAD_CONST            0  (None)
              24   RETURN_VALUE
```

The preceding opcode highlighted contains the following information:

The **STORE_FAST** opcode assigns the value from the range function to the variable

The **LOAD_FAST** opcode adds the value to the top of the stack pointer to be used by the **YIELD_VALUE** opcode.

The following code block explains the working of the **YIELD_VALUE** opcode:

Python/ceval.c (Line no 2082)

```
case TARGET(YIELD_VALUE): {

retval = POP(); // -> 1
...
f->f_stacktop = stack_pointer;
goto exit_yielding; // -> 2
}
```

Python/ceval.c (Line no 3793)

```
exit_yielding:
....

/* pop frame */
exit_eval_frame:
if (PyDTrace_FUNCTION_RETURN_ENABLED())
dtrace_function_return(f);
Py_LeaveRecursiveCall();
f->f_executing = 0; // -> 3
tstate->frame = f->f_back; // -> 4
return _Py_CheckFunctionResult(NULL, retval,
"PyEval_EvalFrameEx"); // -> 5
```

}

The code insights are as

Pop the value from the top of the stack that was loaded previously using the **LOAD_FAST** opcode and assign it to the variable **retval** that is the value returned from the interpreter.

Go to completion of yield is similar to exiting a function.

Mark the frame as not being executed currently.

Assign the frame to be executed next as the frame behind the generator in the function frame stack.

Return the value **retval** from the interpreter.

At this stage, the function is not executed completely, but the frame is held in execution at a particular opcode. When the next function is called again on the same generator instance, the interpreter uses this saved state to start execution at exactly the same opcode until it either yields a value again or raises the **StopIterationException,** which marks the generator instance as completed.

Conclusion

This chapter has covered the internals of function creation and storage in the **PyFunctionObject,** which contains the code, name of the function, and other information, which is then added to a stack frame when calling the function. The storage structure of the **PyFunctionObject** and the **PyFrameObject,** which executes the function call, was also executed.

The creation of the generator and how it differs from a normal function and the flags added at compile time helps create the generator instance. The structure of the **PyGenObject,** which contains a pointer to the stack frame, that is, stopped when yielding and continues back using the saved state. The implementation of the **next** function and how it resumes the execution of the generator instance.

The next chapter covers the basics of memory management in Python as **arenas,** which are pre-allocated pools of memory, and how pools are carved from it on a per request basis. The structure of the pool table and how objects get memory from these pools are created for a particular size of memory is also covered.

CHAPTER 6

Memory Management

In the previous chapter, we covered functions and generators and the similarities and differences between them. Function calls and arrangement of function frames in the thread stack were also covered.

Understanding the memory management of a system can help developers resourcefully choose data management to develop optimized programs. For example, Python requests memory greater than *512 bytes* from the operating system, whereas anything lesser is handled using pre-allocated chunks. This can helps choose data structures sizes efficiently to avoid the higher time required to fetch memory chunks from the operating system. In some cases, the degradation in program performance may be due to frequent memory allocations from the operating system.

Structure

In this chapter, we will cover the following topics:

Memory management overview

Arenas

Arena memory management

Arena memory allocation

Arena memory deallocation

Memory pools

Structure of a memory pool

Memory allocation for objects

Pool table

Allocation lesser than **SMALL_REQUEST_THRESHOLD**

Objective

After studying this chapter, you will be able to understand the memory management layout of Python using the lifecycle of arena objects, the division of arenas into memory pools, and allocating user memory based on the requested memory sizes from predefined chunk pools. You will also learn about the memory allocation for sizes lesser than or greater than the prescribed threshold.

Memory management overview

Memory leaks are common in programs where the developer does not free memory after usage. Such programming errors might lead to severe complications in machine critical programs such as healthcare operating systems and satellites. Programming languages such as Python handle memory allocation on behalf of the programmer and take the responsibility of freeing it when not used. Python creates an internal memory map called **arenas** to save frequent requests to the operating system for memory. This section primarily covers the internal memory management layout on the division of memory into arenas and its division into pools.

The primary unit of the Python memory chunk is called an Arenas are pre-allocated memory chunks the size of which can be configured.

Code block explaining the size of an arena is as follows:

Objects/obmalloc.c (Line no 908)
#define ARENA_SIZE (256 << 10) /* 256KB */

The code block indicates that the default size of an arena is **256** Further, each arena is divided into pools, each of **4 KB** in size:

Objects/obmalloc.c (Line no 908)

```
#define POOL_SIZE
SYSTEM_PAGE_SIZE            /* must be 2^N */
#define POOL_SIZE_MASK              SYSTEM_PAGE_SIZE_MASK
```

Objects/obmalloc.c (Line no 883)

```
#define SYSTEM_PAGE_SIZE            (4 * 1024)
```

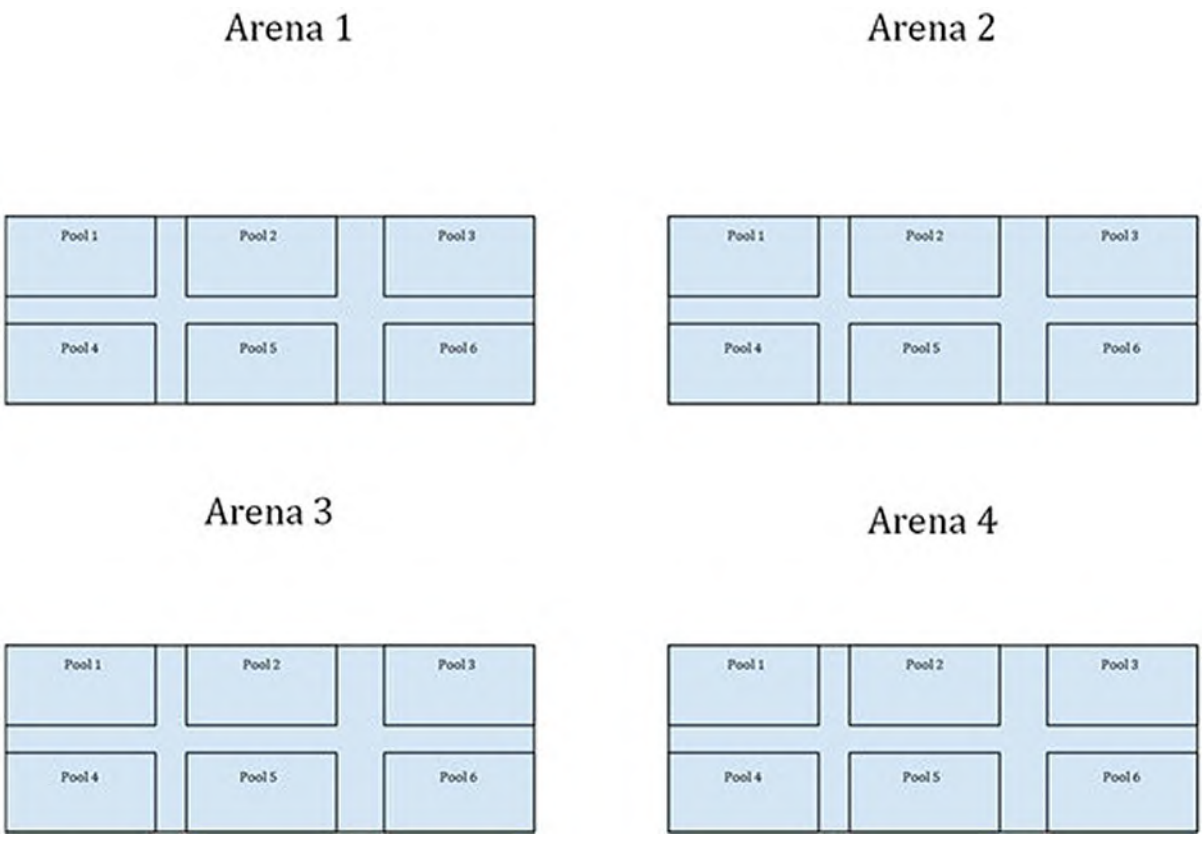

Figure 6.1: *Python memory management as arenas divided into pools*

The pools are further divided into classes based on the size of the requested memory. The size from 1 to 8 bytes is categorized as class 1, 8–16 bytes as class 2, and so on until 504–512 bytes,

which is class 64. Any memory requested for more than 512 bytes is provided through system calls.

Arenas

An arena is a wrapper to a pointer to a large memory location. Pools are virtual memory segments within an arena and are carved out on requirements for allocation:

Objects/obmalloc.c (Line no 951)

```
struct arena_object {
/* The address of the arena, as returned by malloc.  Note that 0
* will never be returned by a successful malloc, and is used
* here to mark an arena_object that doesn't correspond to an
* allocated arena.
*/
uintptr_t address; // -> 1

/* Pool-aligned pointer to the next pool to be carved off. */
block* pool_address; // -> 2

/* The number of available pools in the arena:  free pools + never-
* allocated pools.
*/
uint nfreepools;

/* The total number of pools in the arena, whether or not available. */
```

uint ntotalpools;

/* Singly-linked list of available pools. */
struct pool_header* freepools; // -> 3

....
struct arena_object* nextarena;
struct arena_object* prevarena; // -> 4
};

Code insights are as

The allocated memory is assigned to the address pointer.

The current pool from which memory can be provided from the arena.

The pointer to all the free pools that have memory unassigned in the arena.

The pointers to the doubly linked list of the unassigned arenas.

Arena memory management

The raw memory allocator of Python creates arenas of size 256 KB using system calls like **mmap** on Unix and UNIX-based systems. Covering the entire process of memory allocation and freeing for arenas though interesting, is beyond the scope of this book. This section will focus only on the basics of memory allocation and deallocation to arenas in UNIX/UNIX-like systems while leaving the rest to the curiosity of the readers.

For more information on the allocation of arenas, the creation of the usable and unused arena lists refers to the developer documentation in the file

Arena memory allocation

The following code block shows memory allocation for arenas:

Objects/obmalloc.c (line no 1285) -> 1

```
...
address = _PyObject_Arena.alloc(_PyObject_Arena.ctx, ARENA_SIZE);
...
```

Objects/obmalloc.c (line no 434) -> 2

```
static PyObjectArenaAllocator _PyObject_Arena = {NULL,
...
#elif defined(ARENAS_USE_MMAP)
_PyObject_ArenaMmap, _PyObject_ArenaMunmap
...
};
```

Objects/obmalloc.c (line no 146) -> 3

```
static void *
_PyObject_ArenaMmap(void *ctx, size_t size)
{
void *ptr;
ptr = mmap(NULL, size, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0); // -> 3
...
return ptr;
```

}

Code insights are as

Arenas requests memory using a special memory allocator called the

On UNIX / UNIX-like systems, this is routed to the system call **mmap** that requests for memory from the operating system. Other systems rely on **malloc** and Microsoft Windows OS-based systems that use the **VirtualAlloc** system call.

Request for size bytes from the operating system, which is both readable and writable from. The requested memory is private to this assigned process, and the anonymous flag indicates that the data will always remain in memory and not backed by any file as indicated by the next argument which is the file descriptor.

Arena memory deallocation

An arena is cleared from memory when all pools return to the free state. This method is called during object deallocation on a pool in an arena where all pools are free:

Objects/obmalloc.c (Line no 1802) **-> 1**

```
_PyObject_Arena.free(_PyObject_Arena.ctx,
(void *)ao->address, ARENA_SIZE);
```

Objects/obmalloc.c (line no 434) **-> 2**

```
static PyObjectArenaAllocator _PyObject_Arena = {NULL,
...
#elif defined(ARENAS_USE_MMAP)
_PyObject_ArenaMmap, _PyObject_ArenaMunmap
...
};
```

Objects/obmalloc.c (line no 158) **-> 3**

```
static void _PyObject_ArenaMunmap(void *ctx, void *ptr, size_t
size)
{
munmap(ptr, size);
```

}

Code insights are as

Arenas free its memory using the special memory deallocator called the

On UNIX/UNIX-like systems, this is routed to the system call **munmap,** which frees up the memory directly from the operating system. Other systems either use the free C standard library function, whereas Microsoft Windows OS-based systems use the **VirtualFree** system call.

Return the used memory from the process to the operating system using the **munmap** system call.

Memory_pools

An arena (default 256 KB) is divided into pools of 4 KB memory. The memory required for a Python object is carved from pools based on the size requirements of the objects. The sizes between 1 byte to 8 bytes are referred to as **class** between 9 and 18 bytes are called **class** and so on. As stated previously, the maximum requested size of memory is 512 bytes, referred to as **class**

The following code block describes the division of arenas to pools:

Objects/obmalloc.c (Line no 919)
#define MAX_POOLS_IN_ARENA (ARENA_SIZE / POOL_SIZE) -> 1

Code insight is as

The number of pools in an arena can be a maximum of *256 / 4 = 64* pools, each of 4 KB in size.

Structure of a memory pool

A memory pool is a virtual segment of memory with an arena. A pool is assigned to a size class, and a pointer always points at the head of the pool of free memory to assign the next block of memory to the request:

Objects/obmalloc.c (Line no 933)

```
struct pool_header {
union {block *_padding;
uint count;} ref;          /* number of allocated blocks    */ -> 1
block *freeblock;                    /* pool's free list head          */
-> 2
struct pool_header *nextpool;  /* next pool of this size class  */
-> 3
struct pool_header *prevpool;  /* previous pool
""            */ -> 4
uint arenaindex;                     /* index into arenas of base adr
*/ -> 5
uint szidx;                          /* block size class
index          */ -> 6
uint nextoffset;                  /* bytes to virgin block         */
uint maxnextoffset;               /* largest valid nextoffset      */
};
```

Code insights are as

The number of allocated blocks in the pool.

A free block pointer points to an 8-bit address of an integer (small int). Every time memory is requested for an object, the **freeblock** pointer moves by the block class size, and the previous value of the **freeblock** pointer is assigned to the requesting allocation. We will cover this in the coming subsection on used pools and allocation.

The **nextpool** pointer points to the pool of the same size class. Once the memory allocation is completed on the pool, the allocation is started in the **nextpool** of the size class, and the current pool is unlinked for allocations.

The **prevpool** pointer points to the pool of the same size class.

The **arenaindex** indicates the arena number in which the current pool resides.

The **szidx** indicates the block class size.

Division of arena into memory pools internally storing objects of different sizes.

Memory allocation for objects

When an object requests for memory from the Python memory allocator for any size, it is rounded off to the next higher power of When an object requests *11* the allocator allocates This is done to support easier division of the pool memory and ensure byte alignments of all objects.

Pool table

The pool table contains the link to all pools of a particular size class. Each element in the pool table contains a pointer to a doubly-linked list to all pools in the size class. Pools with free memory are added to the table for allocation and removed once the pool is completely allocated:

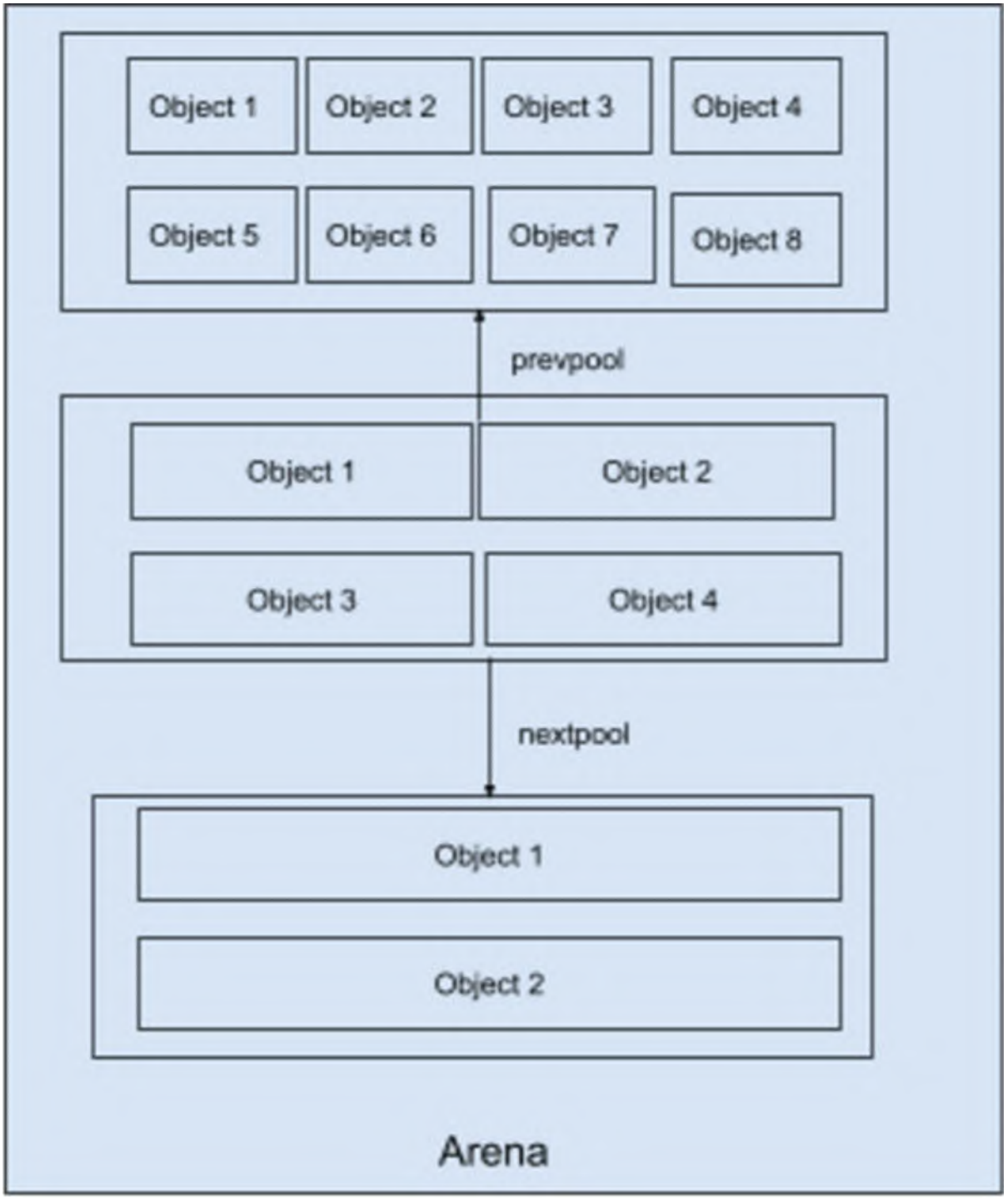

Figure 6.2: *Layout of Objects within pools of different sizes in the arena*

The following code block depicts the structure of the **pool** table:

Objects/obmalloc.c (Line no 1102)

```
#define PTA(x)   ((poolp)((uint8_t *)&(usedpools[2*(x)]) - 2*sizeof(block *)))
#define PT(x)    PTA(x), PTA(x)

static poolp usedpools[2 * ((NB_SMALL_SIZE_CLASSES + 7) / 8) * 8] = {
PT(0), PT(1), PT(2), PT(3), PT(4), PT(5), PT(6), PT(7)
#if NB_SMALL_SIZE_CLASSES > 8
, PT(8), PT(9), PT(10), PT(11), PT(12), PT(13), PT(14), PT(15)
#if NB_SMALL_SIZE_CLASSES > 16
, PT(16), PT(17), PT(18), PT(19), PT(20), PT(21), PT(22), PT(23)
#if NB_SMALL_SIZE_CLASSES > 24
, PT(24), PT(25), PT(26), PT(27), PT(28), PT(29), PT(30), PT(31)
#if NB_SMALL_SIZE_CLASSES > 32
, PT(32), PT(33), PT(34), PT(35), PT(36), PT(37), PT(38), PT(39)
....
#error "NB_SMALL_SIZE_CLASSES should be less than 64"
....
};
```

Code insight is as

Objects/obmalloc.c (line no 1002)

```
/* Python Documentation Starts */
```

Pool table -- headed, circular, doubly-linked lists of partially used pools.

For an index i, usedpools[i+i] is the header for a list of

all partially used pools holding small blocks with "size class idx" i. So usedpools[0] corresponds to blocks of size 8, usedpools[2] to blocks of size 16, and so on: index 2*i <-> blocks of size (i+1)<

Pools are carved off an arena's highwater mark (an arena_object's pool_address member) as needed.

/* Python Documentation Ends */

A memory pool will be in one of the following states:

Pool contains space for allocation and has been allocated for objects at least once.

All blocks in the pool are completely allocated, and the pool is removed from the **usedpool** linked list as no space is available for allocation.

All blocks in the pool are free, and the space is added back to the head of free pools in the **usedpool** linked list.

Allocation lesser than SMALL_REQUEST_THRESHOLD

Memory allocation to objects requesting memory from the python memory allocator calls the function **pymalloc_alloc,** which handles creating arenas when unavailable, carving pools of memory from arenas when no free pools are available, and allocation of memory blocks. Covering this function in depth is beyond the scope of this book; however, the highlights are explained as follows in brief:

Objects/obmalloc.c (Line no 1424)

```
static void* pymalloc_alloc(void *ctx, size_t nbytes) // -> 1
{
....
if (nbytes > SMALL_REQUEST_THRESHOLD) {   // -> 2
return NULL;
}
....
size = (uint)(nbytes - 1) >> ALIGNMENT_SHIFT; // -> 3
pool = usedpools[size + size]; // -> 4

if (pool != pool->nextpool) {
...
++pool->ref.count;
bp = pool->freeblock;     // -> 5
assert(bp != NULL);
```

```
if ((pool->freeblock = *(block **)bp) != NULL) {
goto success;
}

if (pool->nextoffset <= pool->maxnextoffset) {// -> 6
/* There is room for another block. */

pool->freeblock = (block*)pool + pool->nextoffset;
pool->nextoffset += INDEX2SIZE(size);
*(block **)(pool->freeblock) = NULL;
goto success;
}

/* Pool is full, unlink from used pools. */  // -> 7
next = pool->nextpool;
pool = pool->prevpool;
next->prevpool = pool;
pool->nextpool = next;
goto success;
}


...
if (usable_arenas == NULL) {
/* No arena has a free pool:  allocate a new arena. */
....
usable_arenas = new_arena(); // -> 8
...
}
....

init_pool:
```

```
/* Frontlink to used pools. */
next = usedpools[size + size]; /* == prev */
pool->nextpool = next;
pool->prevpool = next;     // -> 10
next->nextpool = pool;
next->prevpool = pool;
pool->ref.count = 1;
....



/* Carve off a new pool. */
....
pool = (poolp)usable_arenas->pool_address; // -> 9
assert((block*)pool <= (block*)usable_arenas->address +
ARENA_SIZE - POOL_SIZE);
pool->arenaindex = (uint)(usable_arenas - arenas);
...
goto init_pool:
success:
assert(bp != NULL);
return (void *)bp; // -> 11
....
```

Code insights are as

The **pymalloc_alloc** function takes two parameters, the pointer to debugging context if in debug mode and the requested memory size.

If the request size is greater than the threshold return **NULL** indicating the allocator to request memory using malloc.

Align the size required to the next higher power of Example: If the request is for *5 bytes*, the allocator returns *8*

Fetch the pool corresponding to the requested memory size class.

Obtain the block of memory from the pool. The **freeblock** pointer holds the current block to be assigned in the pool.

Forward the **freeblock** pointer to the next block in the pool, which can be assigned to the next allocation for the size class.

Unlink the pool from the pool table once full. The pointer is removed from the doubly-linked list for the requested size class.

When all the arenas are consumed by allocations, create a new arena, and start assigning memory from it.

Carve a new pool from an arena when none are found for the requested size class.

Link the created pool into the pool table using the forward and reverse pointers in the pool.

Return the allocated memory block for usage to the requesting object.

Conclusion

This chapter covered the details about Python memory management for objects following a particular size threshold. The memory manager is divided into arenas that are pre-allocated using the operating system calls. Each arena is further split into pools for a particular size class, depending on the request. We also studied the pool table and how it contains a linked list of all pools for a size class. Finally, we covered memory allocation, creation of arenas, and creation of memory pools from arenas.

The upcoming chapter covers the structure of the interpreter and the different opcodes. We will also cover the implementation of a few key opcodes and the design of the stack-based interpreter.

Reader exercises

Analyze the workings of the **pymalloc_free** function to understand how the memory chunks are returned back to the allocated pools.

Analyze the flow when the requested memory is greater than

CHAPTER 7

Interpreter and Opcodes

In the previous chapter, we covered memory allocation and how Python distributes its memory into arenas and sub-divides them into **pools** . The pools are further divided into blocks of a particular size class.

The Python interpreter is stack-based and executes the opcodes generated by the compiler with the arguments given by the user. Understanding interpreter execution is key to understanding concepts, such as thread state, the function stack frame, creation and management of types, iterables, functions, classes, and so on. Threads acquire and relinquish the GIL in the interpreter loop while executing instructions. Generated by the compiler, opcodes define the basic possible functionalities in Python. This chapter covers the definition of opcodes and the execution of the same by the interpreter.

Structure

In this chapter, we will cover the following topics:

Opcodes

Python interpreter stack opcodes

Interpreter stack

Stack operation opcodes

Numerical operation opcodes

Matrix operation opcodes

Iterable opcodes

Looping opcodes

Branching opcodes

Implementation of the interpreter

Opcode prediction

Opcode dispatching and the GIL

Dispatch using computed go-tos

Dispatch without computed go-tos

Signal handling

Initializing signal handlers

Listening to signals

Signals and the interpreter

Objectives

After reading this chapter, you will be able to understand the internal workings of the interpreter and the execution optimizations using branching dispatches.

Opcodes

Opcodes define the functional set of a programming language. Each language defines its master set of opcodes and has a compiler to generate it. Interpreted languages such as Python contain a virtual machine (also known as to execute the generated opcodes, whereas compiled languages such as C++ generate the machine-specific opcodes to be executed on the CPU. Covering the entire set of opcodes of Python is beyond the scope of this book, but an overview of opcode types is described in the following sections. Python 3.8 contains *121 opcodes*, each performing a specific function.

The following code block defines a few opcodes from the opcode set:

Include/opcode.h (Line no 10)

```
...
#define POP_TOP                   1
#define ROT_TWO                   2
#define ROT_THREE                 3
...
```

The preceding code block explains the following:

Each opcode has a name and a unique identifier to identify its execution by the interpreter.

Python interpreter stack opcodes

Understanding the structure and management of the Python interpreter stack forms the basis for all opcodes that rely on it to manage data in transit.

Interpreter stack

The interpreter stack is a pointer to a stack of which are arranged one on top of the other. The stack being generic in nature, can store objects of all types.

The following code block shows the structure of the interpreter stack frame:

```
Python/ceval.c (Line no 750)
_PyEval_EvalFrameDefault(PyFrameObject *f, int throwflag)
{
#ifdef DXPAIRS
int lastopcode = 0;
#endif
PyObject **stack_pointer;   /* Next free slot in value stack */
.....
```

The highlighted code block shows the interpreter stack as a list of **PyObjects** on which stack operations are performed.

The following code block shows the operations on the interpreter stack frame:

```
Python/ceval.c (Line no 950)
#define STACK_LEVEL()     ((int)(stack_pointer - f->f_valuestack))
```

The stack level is used while unwinding the stack frames during exception handling:

```
#define EMPTY()                 (STACK_LEVEL() == 0)
```

The macro checks the size of the stack and indicates if it is empty.

```
#define TOP()                   (stack_pointer[-1])
```

The macro returns the value at the top of the stack. The macro is used to read the value and retain the data on the top of the stack:

```
#define SECOND()                (stack_pointer[-2])
#define THIRD()                 (stack_pointer[-3])
#define FOURTH()                (stack_pointer[-4])
```

The macros return the value at the second, third, and fourth positions from the top of the stack. The macro is used to read the value and also retain the data at the respective positions:

```
#define PEEK(n)                 (stack_pointer[-(n)])
```

The macros return the value at the position from the top of the stack. The macro is used to read the value and also retain the data at the respective position:

```
#define SET_TOP(v)              (stack_pointer[-1] = (v))
```

The macros set the value at the top of the stack. The macro is used when the value must be replaced instead of placing it above the current value at the top of the stack:

```
#define SET_SECOND(v)        (stack_pointer[-2] = (v))
#define SET_THIRD(v)         (stack_pointer[-3] = (v))
#define SET_FOURTH(v)        (stack_pointer[-4] = (v))
```

The macros set the value at the second, third, and fourth positions of the stack. The macro is used when the value must be replaced instead of placing it above the mentioned position from the top of the stack:

```
#define SET_VALUE(n, v)      (stack_pointer[-(n)] = (v))
```

The macros set the value at the position from the top of the stack. The macro is used when the value must be replaced instead of placing it above the mentioned position from the top of the stack:

```
#define BASIC_STACKADJ(n)  (stack_pointer += n)
#define BASIC_PUSH(v)        (*stack_pointer++ = (v))
#define BASIC_POP()          (*--stack_pointer)
```

The macros set and remove the value at the top of the stack. The macros form the core functioning of the stack frame.

Stack operation opcodes

User programs consume the function stack for operations such as tuple creation, calling a function, creating a function, and so on. The Python opcodes set has a few for operations function that has stack pointer operations:

```
#define LOAD_CONST                  100
```

The **LOAD_CONST** opcode loads a particular constant value to the function stack pointer. The loaded constant value is loaded to be processed by an opcode that precedes the loading value.

The following code block demonstrates the implementation of the **LOAD_CONST** opcode:

Python/ceval.c (Line no 1346)

```
case TARGET(LOAD_CONST): {
PREDICTED(LOAD_CONST);
PyObject *value = GETITEM(consts, oparg); // -> 1
Py_INCREF(value);
PUSH(value);   // -> 2
FAST_DISPATCH();
}
```

Code insights are as

Fetch the value from the constants dictionary created by the compiler.

Push the value onto the top of the stack, the implementation of which has been covered in the previous section.

#define POP_TOP 1

The **POP_TOP** opcode returns the constant value at the top of the function stack pointer and subsequently removes it.

The following code block demonstrates the implementation of the **POP_TOP** opcode:

Python/ceval.c (Line no 1361)

```
case TARGET(POP_TOP): {
PyObject *value = POP(); // -> 1
Py_DECREF(value);
FAST_DISPATCH();
}
```

Code insight is as

Pop the value at the top of the function stack, the implementation of which we have discussed in the previous section.

```
#define ROT_TWO                         2
```

The **ROT_TWO** opcode swaps the places of the top two values in the function stack pointer.

The following code block demonstrates the implementation of the **ROT_TWO** opcodes:

Python/ceval.c (Line no 1367)

```
case TARGET(ROT_TWO): {
PyObject *top = TOP();   // -> 1
PyObject *second = SECOND(); // -> 2
SET_TOP(second); // -> 3
SET_SECOND(top); // -> 4
FAST_DISPATCH();
}
```

Code insights are as

Fetch the value from the top of the stack.

Fetch the second value from the top of the stack.

Set the value at the second position to the top of the stack.

Set the value previously at the top of the stack to the second place from the top of the stack.

The opcodes **ROT_THREE** and **ROT_FOUR** are similar in operation to and hence the implementation is skipped in this section, whereas the readers are highly encouraged to examine the implementation of the same:

#define DUP_TOP 4

The **DUP_TOP** opcode duplicates the value at the top of the stack pointer and places a copy at the top of the stack pointer:

Python/ceval.c (Line no 1397)

```
case TARGET(DUP_TOP): {
PyObject *top = TOP(); // -> 1
Py_INCREF(top);
PUSH(top); // -> 2
FAST_DISPATCH();
}
```

Code insights are as

Fetch the current value at the top of the stack.

Duplicate it by pushing it at the top of the stack.

#define DUP_TOP 4

The **DUP_TOP_TWO** opcode duplicates the top two values at the top of the stack pointer and places a copy of both at the top of the stack pointer:

Python/ceval.c (Line no 1397)

```
case TARGET(DUP_TOP_TWO): {
PyObject *top = TOP(); //  -> 1
PyObject *second = SECOND(); // -> 2
Py_INCREF(top);
Py_INCREF(second);

STACK_GROW(2);
SET_TOP(top);   // -> 3
SET_SECOND(second); // -> 4
FAST_DISPATCH();
}
```

Code insights are as

Fetch the current value at the top of the stack as the variable top.

Fetch the second value at the top of the stack as the variable second.

Duplicate top by pushing it to the top of the stack.

Duplicate second by pushing it to the second value from the top of the stack.

Numerical operation opcodes

Numerical operations are among the most common operations performed in any program. This section covers arithmetic operations and also their in-place implementations.

The following code block demonstrates the implementation of **BINARY_ADD** opcode:

```
#define BINARY_ADD      23
```

Python/ceval.c (Line no 1543)

```
case TARGET(BINARY_ADD): {
PyObject *right = POP();
PyObject *left = TOP(); // -> 1
PyObject *sum;
...
if (PyUnicode_CheckExact(left) &&
PyUnicode_CheckExact(right)) {
sum = unicode_concatenate(tstate, left, right, f, next_instr); //->2
...
}
else {
sum = PyNumber_Add(left, right); // -> 3
Py_DECREF(left);
}
```

```
Py_DECREF(right);
SET_TOP(sum);   // -> 4
if (sum == NULL)
goto error;
DISPATCH();
}
```

Code insights are as

Fetch the arguments from the top of the stack.

If both the arguments to the opcode are strings, use the string concatenation operation to compute the joint string.

Call the C function **PyNumber_Add** to compute the sum of the two numerical opcodes. Note, this function also internally can handle the concatenation of lists, sets, and all types that support the + operator.

Set the computed value at the top of the stack.

The following code block demonstrates the implementation of the **BINARY_SUBTRACT** opcode:

```
#define BINARY_SUBTRACT      24
```

Python/ceval.c (Line no 1543)

```
case TARGET(BINARY_SUBTRACT): {
PyObject *right = POP();
PyObject *left = TOP(); // -> 1
PyObject *diff = PyNumber_Subtract(left, right); // -> 2
Py_DECREF(right);
Py_DECREF(left);
SET_TOP(diff); // -> 3
if (diff == NULL)
goto error;
DISPATCH();
}
```

Code insights are as

Fetch the arguments from the top of the stack. These arguments would have typically been added to the stack using the **LOAD_CONST** / **LOAD_NAME** opcodes.

Call the C function **PyNumber_Subtract** to compute the difference between the two numerical arguments. Note, this function also internally can handle the difference of sets and types that support the **-** operator.

Load the difference to the top of the stack.

The implementation of the opcodes **BINARY_DIVIDE** is similar in operation to the and hence is skipped in this section, whereas it is encouraged to examine the implementation of the same.

The following code block demonstrates the implementation of **INPLACE_ADD** opcode:

```
#define INPLACE_ADD                    55

case TARGET(INPLACE_ADD): {
PyObject *right = POP();
PyObject *left = TOP(); // -> 1
PyObject *sum;
if (PyUnicode_CheckExact(left) && PyUnicode_CheckExact(right)) {
sum = unicode_concatenate(tstate, left, right, f, next_instr); // -> 2
}
else {
sum = PyNumber_InPlaceAdd(left, right); // -> 3
Py_DECREF(left);
}
Py_DECREF(right);

SET_TOP(sum); // -> 4

if (sum == NULL)
goto error;
DISPATCH();
}
```

Code insights are as

Fetch the arguments from the top of the stack. These arguments would have typically been added to the stack using the opcodes.

The argument left will be the same to which the sum of the numbers will be assigned.

If both the arguments are of the **string** type, append the strings to a concatenated string.

The C function **PyNumber_InPlaceAdd** is to compute the sum of the two numerical arguments. Note, this function also internally can handle the in-place sum of lists, sets, and types that support the **-** operator.

Load the sum to the top of the stack.

The following code block demonstrates the implementation of the **INPLACE_SUBTRACT** opcode:

```
#define INPLACE_SUBTRACT        56

case TARGET(INPLACE_SUBTRACT): {
PyObject *right = POP();
PyObject *left = TOP(); // -> 1
PyObject *diff = PyNumber_InPlaceSubtract(left, right); // -> 2
Py_DECREF(left);
Py_DECREF(right);
SET_TOP(diff); // -> 3

if (diff == NULL)
goto error;
DISPATCH();}
```

Code insights are as

Fetch the arguments from the top of the stack. These arguments would have typically been added to the stack using the opcodes. The argument left will be the same to which the difference of the numbers will be assigned.

The C function **PyNumber_InPlaceSubtract** is to compute the difference between the two numerical arguments. Note, this function also internally can handle the in-place sum of sets and types that support the **-** operator.

Load the difference to the top of the stack.

The implementation of the opcodes **INPLACE_DIVIDE** is similar in operation to the and hence is skipped in this section, whereas it is encouraged to examine the implementation of the same.

Matrix operation opcodes

Python implements opcodes to perform matrix multiplication operations that are rarely used by programmers. This section covers the implementation of the opcodes.

The following code block demonstrates the implementation of the **BINARY_MATRIX_MULTIPLY** opcode:

```
#define BINARY_MATRIX_MULTIPLY     16

case TARGET(BINARY_MATRIX_MULTIPLY): {
PyObject *right = POP();
PyObject *left = TOP(); // -> 1
PyObject *res = PyNumber_MatrixMultiply(left, right); // -> 2
Py_DECREF(left);
Py_DECREF(right);
SET_TOP(res);   // -> 3
if (res == NULL)
goto error;
DISPATCH();
}
```

Code insights are as

Fetch the matrices from the top of the stack. These matrices would have typically been added to the stack using the opcodes.

The matrix left will be the same to which the difference of the numbers will be assigned.

The C function **PyNumber_MatrixMultiply** is to compute the matrix multiplied by the two numerical arguments.

Load the computed matrix at the top of the stack.

The implementation of the **INPLACE_MATRIX_MULTIPLY** being similar to the **BINARY_MATRIX_MULTIPLY** is left to the interest of the reader.

Iterable opcodes

This section covers the creation, iteration, insertion of elements, and accessing of elements in iterable types like and Python uses specific opcodes to facilitate the management of the iterable types.

The following code block demonstrates the implementation of the **BUILD_LIST** opcode:

```
#define     BUILD_LIST                  103

case TARGET(BUILD_LIST): {
PyObject *list =  PyList_New(oparg); // -> 1
if (list == NULL)
goto error;
while (--oparg >= 0) {  // -> 2
PyObject *item = POP();  // -> 3
PyList_SET_ITEM(list, oparg, item); // -> 4
}
PUSH(list);
DISPATCH();
}
```

Code insights are as

Build a new list using the constructor function **PyList_New** of size

Traverse the list in the reverse direction.

Pop the element at the top of the stack.

Add the element to the list from the tail end.

The following code block demonstrates the implementation of the **BUILD_TUPLE** opcode:

```
#define     BUILD_TUPLE                 104

case TARGET(BUILD_TUPLE): {

PyObject *tup = PyTuple_New(oparg); // -> 1
if (tup == NULL)
goto error;
while (--oparg >= 0) {// -> 2
PyObject *item = POP(); // -> 3
PyTuple_SET_ITEM(tup, oparg, item); // -> 4
}
....
}
```

Code insights are as

Build a new tuple using the constructor function **PyTuple_New** of size

Traverse the tuple in the reverse direction.

Pop the element at the top of the stack.

Add the element into the tuple from the tail end.

The following code block demonstrates the implementation of the **BUILD_MAP** opcode:

```
case TARGET(BUILD_MAP): {
Py_ssize_t i;
PyObject *map = _PyDict_NewPresized((Py_ssize_t)oparg); // -> 1
if (map == NULL)
goto error;
for (i = oparg; i > 0; i--) {
int err;
PyObject *key = PEEK(2*i);
PyObject *value = PEEK(2*i - 1); // -> 2
err = PyDict_SetItem(map, key, value); // -> 3
....

}

....
}
```

Code insight are as

Build a new dictionary using the constructor function **_PyDict_NewPresized** of size

Fetch the key and value from the value stack.

Add the value to the dictionary at the key.

The following code block demonstrates the functioning of the **BINARY_SUBSCR** opcode, which handles the accessing of elements using the index in lists, tuples, and sets while using the key that can be any hashable value in dictionaries:

Python/ceval.c (Line no 1581)

```
#define BINARY_SUBSCR                 25

case TARGET(BINARY_SUBSCR): {
PyObject *sub = POP();          // -> 1
PyObject *container = TOP();          // -> 2
PyObject *res = PyObject_GetItem(container, sub); // -> 3
....
DISPATCH();
}
```

Code insights are as

Fetch the index of the container from the top of the stack.

Fetch the container from the function value stack.

Fetch the value at the specified index. The **PyObject_GetItem** function handles the fetching of the value from the list, tuple, set, and dictionaries. The implementation of the type-specific handlers has been covered in *Chapters 3 on Iterable Sequence Objects* and *Chapter 4 on Sets and*

The following code block demonstrates the functioning of the **LIST_APPEND** opcode, which handles the appending of elements while constructing lists using comprehensions:

Python/ceval.c (line no 1653)

```
#define LIST_APPEND                    145
case TARGET(LIST_APPEND): {
PyObject *v = POP(); // -> 1
PyObject *list = PEEK(oparg); // -> 2
int err;
err = PyList_Append(list, v); // -> 3
....
}
```

Code insights are as

Fetch the value to be appended to the end of the list from the top of the stack.

Fetch the container to which the element has to be appended.

Append the value to the list.

The following code block demonstrates the functioning of the **SET_ADD** opcode, which handles the adding of elements, whereas constructing sets using comprehensions:

Python/ceval.c (Line no 1665)

```
#define SET_ADD                         146

case TARGET(SET_ADD): {
PyObject *v = POP(); // -> 1
PyObject *set = PEEK(oparg); // -> 2
int err;

err = PySet_Add(set, v); // -> 3
....
}
```

Code insights are as

Fetch the value to be added to the set from the top of the stack.

Fetch the set to which the element has to be added to.

Add the value to the set.

The following code block demonstrates the functioning of the **MAP_ADD** opcode, which handles the adding of elements using key and value, whereas constructing dictionaries using comprehensions:

Python/ceval.c (Line no 1665)

```
#define MAP_ADD                 146
case TARGET(MAP_ADD): {
PyObject *value = TOP();    // -> 1
PyObject *key = SECOND();   // -> 2
PyObject *map;
int err;
STACK_SHRINK(2);
map = PEEK(oparg);              // -> dict */
assert(PyDict_CheckExact(map));
err = PyDict_SetItem(map, key, value);  /* map[key] = value */ //
-> 4
...
DISPATCH();
}
```

Code insights are as

Fetch the value to be added to the map from the top of the stack.

Fetch the key to be added to the map, that is, the second element from the top of the stack.

Fetch the map to which the element has to be added at the location pointed by the key.

Add the value to the map.

Looping opcodes

The **for** loop is the most popular construct in Python handles the iteration of built-in iterable types and custom types that implement the **__iter__ and __next__** functions.

Iteration involves two segments, which are as follows:

Fetching of the iterator from the **iterable** object.

Using the **iterable** object to access the elements in the **iterable** object.

The following code block explains the functioning of the **GET_ITER** opcode, which fetches the iterator from the **iterable** type:

Include/opcode.h (Line no 50)

```
#define GET_ITER     68
```

```
case TARGET(GET_ITER): {
PyObject *iterable = TOP(); // -> 1
PyObject *iter = PyObject_GetIter(iterable); // -> 2
....
}
```

Code insights are as

Fetch the **iterable** object from the top of the stack.

The **PyObject_GetIter** function handles the fetching of the iterator of an **iterable** object.

The following code block explains the functioning of the **FOR_ITER** opcode, which iterates the values in the **iterable** type:

Include/opcode.h (Line no 50)

```
#define FOR_ITER     68
```

```
case TARGET(FOR_ITER): {
PREDICTED(FOR_ITER);
/* before: [iter]; after: [iter, iter()] *or* [] */

PyObject *iter = TOP(); // -> 1
PyObject *next = (*iter->ob_type->tp_iternext)(iter); // -> 2
if (next != NULL) {
PUSH(next); // -> 3
....
}
if (_PyErr_Occurred(tstate)) {
if (!_PyErr_ExceptionMatches(tstate, PyExc_StopIteration)) {// -> 3
goto error;
}
....
}
```

Code insights are as

Fetch the iterator from the top of the stack.

Fetch the next value from the **iterable** type using the iterator.

Stop iteration when the iterator raises the **PyExc_StopIteration** exception.

Branching opcodes

Branching opcodes deal with operations related to switching control flows for conditional operators, which are the and **else** The idea is to move the execution of the interpreter code to the associated byte number to which the current execution flow is operating.

An example is as

```
for i in range(0, 20):
if i == 4:
break
```

Opcode for the preceding program:

```
3           0 LOAD_NAME                0 (range)
2 LOAD_CONST               0 (0)
4 LOAD_CONST               1 (20)
6 CALL_FUNCTION            2
8 GET_ITER
>>   10 FOR_ITER                16 (to 28) // -> 1
12 STORE_NAME               1 (i) // -> 2
4           14 LOAD_NAME                1 (i)  // -> 3
16 LOAD_CONST               2 (4) // -> 4
18 COMPARE_OP               2 (==) // -> 5
20 POP_JUMP_IF_FALSE       10 // -> 6
```

```
5           22 POP_TOP                    // -> 7
24 JUMP_ABSOLUTE          28 // -> 8
26 JUMP_ABSOLUTE          10  // -> 9
>>    28 LOAD_CONST               3 (None)
30 RETURN_VALUE
```

This section briefly covers the functioning of the opcodes, whereas the subsections will cover the implementation of each of the opcodes in detail:

The byte number of the **FOR_ITER** opcode is

Store the value of the iteration into the variable

Load the value of **i** into the function value stack.

Load the constant value of **4** into the function value stack.

Compare the value of if equal to

If the value is not equal to move to bytecode **10,** which is the opcode

If the value is we jump to bytecode **28,** which is the exit of the program.

The following code block demonstrates the functioning of the **COMPARE_OP** opcode:

#define COMPARE_OP 107

Python/ceval.c (line no 2974)

```
case TARGET(COMPARE_OP): {
PyObject *right = POP(); // -> 1
PyObject *left = TOP();  // -> 2
PyObject *res = cmp_outcome(tstate, oparg, left, right); // -> 3
...
SET_TOP(res);
...
PREDICT(POP_JUMP_IF_FALSE);
PREDICT(POP_JUMP_IF_TRUE);
DISPATCH();
}
```

Python/ceval.c (line no 5066)

```
static PyObject* cmp_outcome(PyThreadState *tstate, int op,
PyObject *v, PyObject *w)
{
int res = 0;
switch (op) {
...
default:
return PyObject_RichCompare(v, w, op); // -> 4
}
v = res ? Py_True : Py_False; // -> 5
```

```
Py_INCREF(v);
return v; // -> 6
}
```

Code insights are as

Pop the value from the top of the stack.

Fetch the value at the current top of the stack.

Compare the values using the **cmp_outcome** function, which takes an operator in the current case is

The **PyObject_RichCompare** function takes two **Pyobjects** and compares their type and value.

Return the value as a **PyBool** object depending on the result of the comparison.

The **cmp_outcome** function also checks for operators such as **not** and **is** Covering the entire implementation of the function is beyond the scope of the book but is highly encouraged.

The following code block demonstrates the functioning of the **POP_JUMP_IF_FALSE** opcode:

```
#define POP_JUMP_IF_FALSE
```

Python/ceval.c (line no 3041)

```
case TARGET(POP_JUMP_IF_FALSE): {
PREDICTED(POP_JUMP_IF_FALSE);
PyObject *cond = POP(); // -> 1
int err;
if (cond == Py_True) {// -> 2
Py_DECREF(cond);
FAST_DISPATCH(); // -> 3
}
if (cond == Py_False) {// -> 4
Py_DECREF(cond);
JUMPTO(oparg); // -> 5
FAST_DISPATCH(); // -> 6
}
....
DISPATCH();
}
```

Code insights are as

Pop the value from the top of the stack, which is the result of the previous comparison operation.

If the result value is we do not have to branch the control flow of execution.

Dispatch the execution to the next opcode being executed.

If the result value is we have to branch the control flow of execution by branching to the bytecode mentioned as an argument to the opcode.

Branch to the bytecode mentioned as result is false. The implementation of the **JUMPTO** macro is covered later in the chapter. Dispatch to the next instruction mentioned in the bytecode.

The following code block demonstrates the functioning of the **JUMP_ABSOLUTE** opcode:

```
case TARGET(JUMP_ABSOLUTE): {
PREDICTED(JUMP_ABSOLUTE);
JUMPTO(oparg);    // -> 1
....
DISPATCH();
}
```

Code insight is as

Jump to the bytecode mentioned as the argument to the opcode.

Implementation of the interpreter

The Python interpreter is a large C function that contains the implementation of all opcodes within a giant **for** loop. The interpreter handles cooperative multithreading using the GIL and also implements opcode dispatching to speed up the execution of the opcodes. This section deals with only the relation between the interpreter and the GIL, although the implementation of the GIL is handled in *Chapter 8 on GIL and multithreading* in detail.

Opcode prediction

The interpreter executes the opcodes one after the other within a giant **for** loop, and in many cases, the assembler can predetermine the next opcode based on the current one. This technique can be used to speed up the execution by using computed executing **go-to** to a predetermined code location stored in a pre-computed branch table. This technique can result in faster code execution on certain CPUs overusing the switch condition within the interpreter.

Code sample explaining the use of opcode dispatching:

```
case TARGET(LIST_APPEND): {
...
err = PyList_Append(list, v); // -> 1
...
PREDICT(JUMP_ABSOLUTE); // -> 2
DISPATCH(); // -> 3
}
```

Code insights are as

The example demonstrates the use of the **PREDICT** and **DISPATCH** opcode in the implementation of the **LIST_APPEND** opcode.

The most frequent opcode posts the execution is the **JUMP_ABSOLUTE** used to return to the initial opcode of the list apprehension.

Dispatch the opcode by jumping to the code block pointed by the pre-computed go-to table.

Python/ceval.c (Line no 929)

```
#if defined(DYNAMIC_EXECUTION_PROFILE) || USE_COMPUTED_GOTOS

#define PREDICT(op)             if (0) goto PRED_##op
#else
#define PREDICT(op) \
do{\
_Py_CODEUNIT word = *next_instr; \ // -> 1
opcode = _Py_OPCODE(word); \ // -> 2
if (opcode == op){ \ /// -> 3
oparg = _Py_OPARG(word); \
next_instr++; \
goto PRED_##op; \ // -> 4
} \
} while(0)
#endif

#define PREDICTED(op)           PRED_##op: // -> 5
```

Python/ceval.c (line no 3139)

```
case TARGET(JUMP_ABSOLUTE): {
```

PREDICTED(JUMP_ABSOLUTE); // -> 6
JUMPTO(oparg);
...
}

Code insights are as

Fetch the next instruction to be executed using the **next_instr** pointer.

Fetch the opcode from the instruction.

Check if the next opcode to be executed is the same as predicted.

If yes, jump to the execution of the predicted opcode, else use the **switch** case in the **for** loop to execute the next opcode.

The macro **PREDICTED** creates the label to jump to using the **goto** statement.

The **PREDICTED(JUMP_ABSOLUTE)** creates the label for the opcode **JUMP_ABSOLUTE** as

The prediction logic is not used in cases when Python is running in a multi-threaded environment as the branch prediction becomes unreliable for the CPU.

Opcode dispatching and the GIL

The Python interpreter handles the GIL and allows threads to execute mutually using cooperative multithreading. Each thread on every opcode cycle with computation drops the GIL and checks for any other thread that accepts the request on the opcode cycle. If there is another request, it switches the context to the other thread and begins executing it until it enters another dispatching cycle where it has to relinquish it to any other thread.

The interpreter uses two forms of dispatching, fast dispatching for opcodes that do not involve computations such as stack operations, NOP, and so on. In this case, the interpreter moves to execute the next opcode and does not drop the GIL. The normal dispatch involves computation such as adding numbers, creating a list, and so on, also handles cooperative multithreading.

The following code block demonstrates the usage and implementation of **FAST_DISPATCH** in opcodes that do not involve computation:

```
case TARGET(LOAD_CONST): {
PREDICTED(LOAD_CONST);
PyObject *value = GETITEM(consts, oparg);
Py_INCREF(value);
PUSH(value);
FAST_DISPATCH(); // Usage of fast dispatch in LOAD_CONST.
```

```
}
```

Dispatch using computed go-tos

Some compilers support the creation of computed which enable the creation of dynamic jump tables, helping avoid the complex **switch** case to execute the opcode. On such compilers that handle it, Python creates the jump targets using the macro the definition of which has been explained as follows:

The following code block demonstrates the implementation of the **FAST_DISPATCH** opcode:

Python/ceval.c (line no 836)

```
#if USE_COMPUTED_GOTOS
/* Import the static jump table */
#include "opcode_targets.h" // -> 1

#define TARGET(op) \
op: \
TARGET_##op // -> 2

#define FAST_DISPATCH() \
{\
if (!_Py_TracingPossible(ceval) && !PyDTrace_LINE_ENABLED()) {\
f->f_lasti = INSTR_OFFSET(); \
NEXTOPARG(); \
goto *opcode_targets[opcode]; \ // -> 3
```

```
} \
goto fast_next_opcode; \ // -> 4
}
```

Python/opcode_targets.h (line no 1)

```
static void *opcode_targets[256] = {
&&_unknown_opcode,
&&TARGET_POP_TOP, // -> 5

&&TARGET_ROT_TWO,
&&TARGET_ROT_THREE,
&&TARGET_DUP_TOP,
...
}
```

Code insights are as

When the usage of computed go-tos is enabled, the file **opcode_targets.h** is included, which contains all the addresses of the opcode targets to jump execution.

Taking the example of it creates the **goto** label **TARGET_LOAD_CONST** the address of which is stored in the which is in the file **opcode_targets.h** explained in the following *point*

Use the address in the jump table to go to the execution of the next opcode.

If tracing is enabled, the interpreter begins the execution of the next opcode, which begins at the label When the execution moves to the interpreter skips the handling of requests for the GIL. Hence opcodes that call **FAST_DISPATCH** do not relinquish the thread, whereas requests to **DISPATCH** may relinquish if any other threads are waiting for the GIL.

The jump targets are defined in the file which contains a giant array of the addresses of the opcodes. The index of the target in the array is the numerical value of the opcode defined in the file For example, the value of the opcode **UNARY_INVERT** is and the index in the array is also

The following code block demonstrates the implementation of the **DISPATCH** opcode:

Python/ceval.c (line no 866)

```
#define DISPATCH() \
{\
if (!_Py_atomic_load_relaxed(eval_breaker)) {\ // -> 1
FAST_DISPATCH(); \ // -> 3
} \
continue; \ // -> 2
}
```

Code insights are as

Check if any pending threads are awaiting the GIL.

If yes, continue to the giant for loop, which handles relinquishing the GIL to the other threads and later continues the execution of the other thread. This will be covered in detail in the upcoming section.

If no other threads have been requested for the GIL, use **FAST_DISPATCH** to execute the next instruction on the current thread.

Dispatch without computed go-tos

On compilers that do not support pre-computed go-tos, the implementation of **FAST_DISPATCH** and **DISPATCH** macros is relatively simpler demonstrated as follows:

Python/ceval.c (line no 875)

```
#define TARGET(op) op // -> 1
#define FAST_DISPATCH() goto fast_next_opcode // -> 2
#define DISPATCH() continue // -> 3
```

Code insights are as

The **TARGET** macro does not expand to **goto** label but expands directly to the **switch** case.

The **FAST_DISPATCH** macro directly skips the execution of the GIL transfer and starts the execution of the next opcode.

The **DISPATCH** macro continues the loop execution and handles GIL transfer if any threads await the execution.

Signal handling

The Python interpreter while executing the opcodes, might receive signals from the operating system or the user that have to be handled by the process on high priority temporarily pausing the execution of the opcodes. Python handles all the received signals on the main thread, whereas they can be received on any executing thread.

Initializing signal handlers

This section covers signal management in Python on Linux-based systems. Covering for all systems is beyond the scope of this book but is recommended for readers working on either Windows/Solaris systems.

The following code block demonstrates the initialization of signal handlers:

Modules/signalmodule.c (line no 104)

```
static volatile struct {
_Py_atomic_int tripped; // -> 1
PyObject *func; // -> 2
} Handlers[NSIG]; // -> 3
```

Code insights are as

The tripped flag for every signal indicates that the signal has been received by the Python process.

The function to handle the signal.

The **Handlers** array of all the signals to be handled by the process.

The following code block demonstrates the assignment of the default handlers for all signals during initialization of the Python process:

Modules/signalmodule.c (line no 458)

```
for (i = 1; i < NSIG; i++) {
void (*t)(int);
t = PyOS_getsig(i);
_Py_atomic_store_relaxed(&Handlers[i].tripped, 0); // -> 1
if (t == SIG_DFL)
Handlers[i].func = DefaultHandler; // -> 2
else if (t == SIG_IGN)

Handlers[i].func = IgnoreHandler; // -> 3
else
Handlers[i].func = Py_None; /* None of our business */ // -> 4
Py_INCREF(Handlers[i].func);
}
```

Code insights are as

Initialize the tripped flag for every signal to indicating that the signal has not yet been received by the Python process.

Set the default handler for the signal

Set the handler **IgnoreHandler** for the signal

The Python process does not handle any other signals and hence sets the value to In this case, the developer needs the program to listen to any signals; the **signal** function from the signal module can be invoked with the signal number.

Listening to signals

The Python process by default does not listen to any user/kernel level signals and waits for the user to initialize the handling. This can be achieved using the **signal** function in the signal module of Python.

The code sample is as follows:

```
def signal_handler_func():
print "Handling the signal"
import signal
signal.signal(signal.SIGALRM, signal_handler_func) # -> 1
```

Code insight is as

Assign the handler function to the signal

The following code block demonstrates the implementation of the **signal** function that assigns the Python function handling a particular signal:

```
static PyObject * signal_signal_impl(PyObject *module, int
signalnum, PyObject *handler) {
PyObject *old_handler;
void (*func)(int);
```

```
...
func = signal_handler; // -> 1
...
if (PyOS_setsig(signalnum, func) == SIG_ERR) {// -> 2
PyErr_SetFromErrno(PyExc_OSError);
return NULL;
}
old_handler = Handlers[signalnum].func;
Py_INCREF(handler);
Handlers[signalnum].func = handler; // -> 3

if (old_handler != NULL)
return old_handler;
else
Py_RETURN_NONE;
}
```

Code insights are as

Set the function to handle the signal calls to the operating system to be the **signal_handler** function.

The **PyOS_setsig** function assigns the **signal_handler** to the signal on UNIX and UNIX-like systems calls the **signal** function.

Set the **signal** handler to be the passed Python function.

Signals and the interpreter

The previous section covered how programmers can add custom handlers for signals. The Python interpreter continues to execute the instructions of the user's program until the asynchronous signal is received by it. As stated in the previous section, Python handles all signals only on the main thread. If the current thread receives the signal, it ignores it and continues execution until the main thread takes over the GIL to execute the signals.

The interpreter continues to execute user instructions until it receives a request to drop the GIL, handle asynchronous events, or there are pending calls to be executed. In this case, it sets a flag called **eval_breaker** to indicate the interpreter to pause execution of the user's program and handle the signals on priority:

Python/ceval.c (line no 130)

```
#define COMPUTE_EVAL_BREAKER(ceval) \
_Py_atomic_store_relaxed(\
&(ceval)->eval_breaker, \
GIL_REQUEST | \ // -> 1
_Py_atomic_load_relaxed(&(ceval)->signals_pending) | \ // -> 1
_Py_atomic_load_relaxed(&(ceval)->pending.calls_to_do) | \ // -> 1
(ceval)->pending.async_exc) // -> 1
```

Code insight is as

Set the **eval_breaker** if there is a GIL drop request, pending signals, or calls, or any asynchronous execution is pending.

The following code block demonstrates the interpreter handling the signals when the **eval_breaker** is set:

```
if (_Py_atomic_load_relaxed(eval_breaker)) {// -> 1
opcode = _Py_OPCODE(*next_instr); // -> 2
if (opcode == SETUP_FINALLY ||
opcode == SETUP_WITH ||
opcode == BEFORE_ASYNC_WITH ||
opcode == YIELD_FROM) {
....
goto fast_next_opcode; // -> 3
}

if (_Py_atomic_load_relaxed(&ceval->signals_pending)) {// -> 4
if (handle_signals(runtime) != 0) {// -> 5
goto error;
}
}
if (_Py_atomic_load_relaxed(&ceval->pending.calls_to_do)) {// -> 6
if (make_pending_calls(runtime) != 0) {// -> 7
goto error;
}
}
```

```
if (_Py_atomic_load_relaxed(&ceval->gil_drop_request)) {// -> 8
/* Give another thread a chance */
if (_PyThreadState_Swap(&runtime->gilstate, NULL) != tstate) {
Py_FatalError("ceval: tstate mix-up");


}
drop_gil(ceval, tstate); // -> 9


take_gil(ceval, tstate); // -> 10


/* Check if we should make a quick exit. */
....


fast_next_opcode: // -> 11
```

Code insights are as

Bypass the execution of the user's opcodes only if the **eval_breaker** has been set.

Fetch the next opcode to check if there is an opcode that requires urgent attention.

If yes, skip the execution of signals, pending calls, and GIL transfer, and begin executing it.

Check if there are any signals pending to be executed.

If yes, execute them.

Check if there are any async calls pending to be executed.

If yes, execute the pending calls.

Check for any threads requesting the GIL.

If yes, drop the GIL currently held by the current thread.

The GIL is taken by the next thread requesting execution.

Continue the execution of the new thread from the opcode it was executing before the previous context switching.

Conclusion

This chapter begins with the definition of Python opcodes and how they are generated by the assembler. All opcodes of Python are declared in the file **opcode.h** with a numerical value assigned to each of them. Stack-based opcodes perform data operations on the function value-stack, such as and so on.

This chapter covers the implementation of most of the basic stack-based opcodes and leaves the rest to the curiosity of the reader. Numerical opcodes perform addition, subtraction, division, modulo, among other operations on numbers. The same opcodes also support operations on types such as lists, sets, and so on, which support numerical operations of + on its type.

Looping opcodes handle iteration of iterable types such as and so on. or other types which support iteration. Iterable opcodes handle creation and the operations on built-in iterable types such as lists, sets, and dictionaries.

Branching opcodes handle branching and switch condition logic in programs. The interpreter is defined as a function with a giant for loop handling the implementation of each of the opcodes.

Opcode prediction can speed up the execution on single-threaded environments where branch targets can be reliably computed. Dispatching helps manage signal handling, pending calls, and

switching thread contexts using the GIL. Python signal handling allows program developers to develop custom handlers for OS/user-generated signals to be handled by the interpreter.

The next chapter covers the implementation of multithreading using **Pthreads** on UNIX and UNIX-based systems and the implementation of GIL to handle the execution of a single thread per Python process.

CHAPTER 8

GIL and Multithreading

In the previous chapter, we covered opcodes, their types, and the internals of the interpreter. Opcodes are classified based on the operation, such as stack-based, numeric, and so on. Interpreter, along with executing the opcodes, handles functionalities such as signal handling and context switching of threads.

Global interpreter lock is one of the most debatable topics in Python, which restricts the maximum number of threads to access the CPU concurrently to be a maximum of There have been discussions in the Python community about its existence and relevance, and results to eliminate it for performance reasons have not been positive, and hence the GIL exists. The GIL was added to Python to disallow simultaneous access to Python variables across threads resulting in erroneous reference count values. The solution was to either add a global lock such as the GIL or add a lock to each variable resulting in slower performance. Hence the GIL was added to minimize the complexity. This chapter covers the creation and life cycle of Python threads and how the GIL controls the execution of threads on the CPU.

Structure

In this chapter, we will cover the following topics:

The GIL

Structure of GIL

Creation and initialization of the GIL

Taking the GIL to access the interpreter

Relinquishing the GIL

Deallocating the GIL

Multithreading with the GIL

Objective

After studying this unit, you will be able to understand the structure and work of the GIL. You will also learn how threads acquire and relinquish the GIL to enable cooperative multithreading in Python.

The GIL

The GIL is one of the most debatable topics in the Python development community. Even though conceptually, it remains simple and is internally a Boolean variable indicating if a thread holds the value or not. The access to this variable is protected by a mutex, which is signaled by a conditional variable

Python implements cooperative multithreading, and hence the thread holding the GIL must be able to relinquish the thread when requested by another thread. A thread requesting to get access waits for the time as mentioned using the default interval.

The following code block demonstrates the value of the default interval:

```
#define DEFAULT_INTERVAL 5000
```

The default interval is set as *5 milliseconds* before placing the request for the GIL.

Structure of GIL

GIL initialization involves creating an instance of the structure **_gil_runtime_state** the definition of which is explained is as follows. The initialization happens at the time of creating the interpreter that begins to execute the Python opcodes:

internal/pycore_gil.h (line no 23)

struct _gil_runtime_state {
/* microseconds (the Python API uses seconds, though) */
unsigned long interval; // -> 1
/* Last PyThreadState holding/having held the GIL. This helps us know whether anyone else was scheduled after we dropped the GIL. */
_Py_atomic_address last_holder; // -> 2
/* Whether the GIL is already taken (-1 if uninitialized). This is atomic because it can be read without any lock taken in ceval.c. */
_Py_atomic_int locked; // -> 3
/* Number of GIL switches since the beginning. */
unsigned long switch_number; // -> 4
/* This condition variable allows one or several threads to wait until the GIL is released. In addition, the mutex also protects the above variables. */
PyCOND_T cond; // -> 5
PyMUTEX_T mutex; // -> 6
...

Code insights are as

The mandatory interval that a thread waits before it has placed a request to the GIL.

The thread ID of the last thread that held access to the GIL.

The GIL Boolean variable, a value greater than 0, indicates a thread holding the lock, else there are no active threads.

Number of GIL switches since initialization.

The conditional variable that waits for the release of the mutex holds access to set the request for the GIL.

The mutex holds access to place requests for the GIL.

Creating and initializing the GIL

The following code block demonstrates the memory allocation to the GIL:

```
static void create_gil(struct _gil_runtime_state *gil) {
MUTEX_INIT(gil->mutex); // -> 1
...
COND_INIT(gil->cond); // -> 2
...
_Py_atomic_store_relaxed(&gil->last_holder, 0); // -> 3
_Py_atomic_store_explicit(&gil->locked, 0,
_Py_memory_order_release); // -> 4
}
```

Code insights are as

Initialize the mutex **gil->mutex** using the macro

Initialize the conditional variable **gil->cond** on which the thread holding the mutex will wait for the GIL to be released.

Initialize the variable **last_holder** to **0,** indicating that no thread has accessed the interpreter using the GIL.

The GIL is initialized to the value

The following code block demonstrates the initialization of the GIL:

Python/ceval.c (line no 644)

```
void _PyEval_Initialize(struct _ceval_runtime_state *state) // -> 1
{
...
_gil_initialize(&state->gil); // -> 2
}
```

Python/ceval_gil.h (line no 93)

```
static void _gil_initialize(struct _gil_runtime_state *gil) {
_Py_atomic_int uninitialized = {-1};
gil->locked = uninitialized; // -> 3
gil->interval = DEFAULT_INTERVAL; // -> 4
}
```

Code insights are as

Function initializing the interpreter state.

As mentioned previously, the GIL is initialized at the time of interpreter initialization.

Initialize the **gil** to **-1,** indicating that no threads are actively holding it at the beginning.

The switching interval is set to *5 milliseconds* as default.

Taking the GIL to access the interpreter

The thread/s that run their code and compete for the CPU request access to the GIL using the function The function uses the mutex **gil->mutex** to provide access to only one thread to take contention of the interpreter.

The following code block demonstrates the interpretation of the **take_gil** function:

```
Python/ceval_gil.h (line no 184)
static void take_gil(struct _ceval_runtime_state *ceval,
PyThreadState *tstate)
{
...
struct _gil_runtime_state *gil = &ceval->gil; // -> 1
...
MUTEX_LOCK(gil->mutex); // -> 2
if (!_Py_atomic_load_relaxed(&gil->locked)) {
goto _ready; // -> 3
}

while (_Py_atomic_load_relaxed(&gil->locked)) {// -> 10
int timed_out = 0;
unsigned long saved_switchnum;
saved_switchnum = gil->switch_number; //
```

```
unsigned long interval = (gil->interval >= 1 ? gil->interval : 1); // -
> 4
COND_TIMED_WAIT(gil->cond, gil->mutex, interval, timed_out); // -
> 5
/* If we timed out and no switch occurred in the meantime, it is
time
to ask the GIL-holding thread to drop it. */

if (timed_out && // -> 6
_Py_atomic_load_relaxed(&gil->locked) && // -> 7
gil->switch_number == saved_switchnum)
{
SET_GIL_DROP_REQUEST(ceval); // -> 8
}
}
_ready:
...
/* We now hold the GIL */
_Py_atomic_store_relaxed(&gil->locked, 1); // -> 9
_Py_ANNOTATE_RWLOCK_ACQUIRED(&gil->locked, /*is_write=*/1);
...
}
```

Code insights are as

The GIL is a part of the interpreter, fetch its address, and store it in the variable **gil** for function scope.

Request for access to the mutex on the current thread to allow single contention for access to the GIL.

If the GIL is not currently accessed by any threads directly, provide access to the requesting thread.

Fetch the interval to wait for the current thread to relinquish the GIL.

Execute a conditional timed wait on the mutex for the thread holding the GIL to free the mutex. This will be covered in-depth in the upcoming section on relinquishing the GIL.

On timeout, if the current thread has not relinquished control to the GIL, place a request to the current thread to immediately stop execution and provide access to the interpreter.

Same as

Place the request to the executing thread to relinquish control to execute instructions.

Take the GIL and begin executing the instructions on the current thread.

Relinquishing the GIL

The thread currently holding access to the GIL must relinquish access for other threads to begin executing the opcodes. In the previous chapter, we have covered in-depth opcode dispatching and how the executing thread in every interpreter cycle checks for threads waiting to access the CPU. This section covers only the part of the holding thread relinquishing control and releasing the mutex for the other threads to place requests to hold the GIL.

The following code block demonstrates the interpretation of the **drop_gil** function:

```
Python/ceval_gil.h (line no 143)
static void drop_gil(struct _ceval_runtime_state *ceval,
PyThreadState *tstate)
{
struct _gil_runtime_state *gil = &ceval->gil; // -> 1
if (!_Py_atomic_load_relaxed(&gil->locked)) {
Py_FatalError("drop_gil: GIL is not locked"); // -> 2
}

...
MUTEX_LOCK(gil->mutex); // -> 3
_Py_atomic_store_relaxed(&gil->locked, 0); // -> 4
COND_SIGNAL(gil->cond); // -> 5
MUTEX_UNLOCK(gil->mutex); // -> 6
```

...
}

Code insights are as

The GIL is a part of the interpreter, fetch its address, and store it in the variable **gil** for function scope.

If a thread tries to drop a GIL without holding the lock, raise a

Lock the mutex to begin dropping the GIL.

Relinquish control of the GIL.

Signal the waiting threads to wake up from the conditional timeout.

Unlock the mutex to relinquish control to other threads to place requests to the GIL.

Deallocating the GIL

Deallocating the GIL releases the mutex, conditional variable back to the operating system, which are key resources that must be released for other processes to take the OS primitives:

Python/ceval_gil.h (line no 120)

```
static void destroy_gil(struct _gil_runtime_state *gil) {
...
COND_FINI(gil->cond);  // -> 1
MUTEX_FINI(gil->mutex);  // -> 2
...
_Py_atomic_store_explicit(&gil->locked, -1, // -> 3
_Py_memory_order_release);
}
```

Code insights are as

Release the conditional variable to the operating system.

Release the mutex to the operating system.

Reset the GIL value to **-1,** indicating the GIL is no longer used.

Multithreading with the GIL

This section covers the basics of how threads acquire and compete for the GIL from the process of creation of threads to acquiring the GIL and relinquishing for other threads to run on the interpreter.

The code sample is as follows:

```
import thread
def print_name(name="Sundar"): // -> 1
print("Hello: {}".format(name))

thread.start_new_thread(print_name, ("Sampath")) // -> 2
```

The function **print_name** accepts a single string parameter name as the argument.

The **start_new_thread** function of the thread module spawns a new thread and passes the string argument **Sampath** as the **name** parameter.

The following code block demonstrates the declaration and definition of the **start_new_thread** method:

Modules/_threadmodule.c (line no 1447)

{"start_new_thread", (PyCFunction)thread_PyThread_start_new_thread,

Modules/_threadmodule.c (line no 1025)

```
static PyObject * thread_PyThread_start_new_thread(PyObject *self, PyObject *fargs)
{
PyObject *func, *args, *keyw = NULL;
struct bootstate *boot;
unsigned long ident;

...
boot = PyMem_NEW(struct bootstate, 1); // -> 1

...
boot->interp = _PyInterpreterState_Get(); // -> 2
boot->func = func; // -> 3
boot->args = args; // -> 4
boot->keyw = keyw; // -> 5
boot->tstate = _PyThreadState_Prealloc(boot->interp); // -> 6
...
PyEval_InitThreads(); /* Start the interpreter's thread-awareness */
// -> 7
ident = PyThread_start_new_thread(t_bootstrap, (void*) boot); // -> 8
...
return PyLong_FromUnsignedLong(ident);
}
```

The following code block demonstrates the declaration of the structure

Modules/_threadmodule.c (line no 1447)

```
struct bootstate {// -> 9
PyInterpreterState *interp;
PyObject *func;
PyObject *args;
PyObject *keyw;
PyThreadState *tstate;
};
```

The following code block demonstrates the function **t_bootstrap,** which initializes the thread and requests for the GIL to begin execution of the opcode of the thread:

Modules/_threadmodule.c (line no 991)

```
static void t_bootstrap(void *boot_raw)

{
struct bootstate *boot = (struct bootstate *) boot_raw;
...
tstate = boot->tstate;
...
PyEval_AcquireThread(tstate); // -> 10
tstate->interp->num_threads++;
res = PyObject_Call(boot->func, boot->args, boot->keyw); // -> 11
```

...
}

Code insights are as

Initialize a new object of the type thread bootstrap, which is a wrapper around the function opcode and the arguments of the executing thread.

Get the reference to the current executing interpreter state.

Reference the function opcode to the bootstrap wrapper.

Reference the arguments to the function of the bootstrap wrapper.

Reference the keyword arguments to the function of the bootstrap wrapper.

Create a new reference to the current thread state to be executed. It includes the information about the thread that including the reference to the frame, exception information, and so on.

Initialize the GIL if not initialized at this stage.

Start the new thread. The creation of the thread depends on the platform where Python is executing. On UNIX-based systems, Python leverages while on DOS systems, it uses the native threading system.

Demonstration of the structure of the bootstrap object.

Acquire the GIL before executing the opcode of the function.

Execute the opcode of the function. Although executing the opcode, relinquish the GIL to other threads in case they place a request for it. This has been covered in-depth in *Chapter 7 on Interpreter and*

Conclusion

The GIL has been one of the key concepts that fascinate Python developers for a long time. The GIL is a Boolean flag indicating if a thread is actively holding it. It is protected by a mutex, and the thread holding the GIL releases it voluntarily when another thread places a request for the same. This chapter covered the structure, creation, and initialization of the GIL. The acquiring and releasing of the GIL were also discussed at length. The example of GIL swapping at the time of thread creation using the **start_new_thread** was also discussed.

CHAPTER 9

Async Python

Many applications on the server side often end up waiting for database, file, or socket operations to be completed before any action can be taken on the data. This results in a lot of idle time for the request thread when waiting for the response. This idle time on the thread can be used to perform other tasks over idling. This popular paradigm shift has led to the development of server-side frameworks such as **Node.js** in JavaScript/ **Netty** in Java and slowly in multiple programming languages using framework support. Python decided to add support within the programming language for async frameworks from *version 3.4* . Although **async frameworks** are becoming very commonly used in server-side applications these days, the paradigm has always been popular in client-side applications and mobile development where the client has to wait for the server to respond before any action can be taken/wait for the user to perform any action both of which are async in nature.

Example

A waiter in the restaurant takes orders and places them into the kitchen for the chef to start cooking. If the waiter has to wait until the chef returns the cooked order, this will result in a lot of time wasted, which the waiter can use to take other orders. In the former system, the waiter would need a lot more waiters to process the orders, while in the latter, the restaurant needs very few of them.

In the example stated, replacing waiters with threads and orders with requests to the server understand that by not waiting for the database/file to return the data, the thread can multiplex across multiple parallel user requests. This chapter delves into understanding the **await** keywords introduced in the language, and the **asyncIO** framework used to develop coroutines and concurrent async applications in Python.

Structure

In this chapter, we will cover the following topics:

Coroutines

Continuing the execution of the coroutine

Async functions

Objective

After studying this unit, you will be able to understand the implementation of **coroutines** internally using generators. You will also learn how **async functions** are internally implemented, similar to coroutines.

Coroutines

Coroutines are functions the execution of which can be paused in between and be continued on the basis of the input provided to the routine. Coroutines can also be chained to move the input from one coroutine to another, the input of one being passed to the other.

The example code demonstrating a simple coroutine:

```
def check_equals(sum, num1):
print("Checking if sum is equal to :{}".format(sum))
try :
while True:
num2 = (yield) // -> 1
if sum == num1 + num2:
print(“The sum of numbers is equal to the provided sum”)
except GeneratorExit:
print("Exit the coroutine")

coro_instance = check_equals(10, 5)   // -> 2
coro_instance.__next__()   // -> 3
coro_instance.send(4) // -> 4
..
coro_instance.close()
18 YIELD_VALUE                          // ->  5
20 STORE_FAST                  2 (num2) // -> 6
```

...

The code insights are as

The **yield** statement is used to pause the execution of the code at the point to wait for user input to be passed to the function at the point of interruption.

Calling the function creates an instance of the coroutine.

The interpretation of the coroutine begins when the function **__next__** is called on the function only.

The function send is called on the instance of the coroutine with the argument to pass the value to the interrupted function.

The value is fetched back to the function using the opcode **YIELD_VALUE** the implementation of which is provided as follows.

The value obtained from the function is saved in the value

The following code block explains the creation of an instance of coroutine:

Python/ceval.c (line no 4044)

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) {// -> 1
...
if (co->co_flags & (CO_GENERATOR | CO_COROUTINE |
CO_ASYNC_GENERATOR)) {
PyObject *gen;
int is_coro = co->co_flags & CO_COROUTINE; // -> 2
...


if (is_coro) {
gen = PyCoro_New(f, name, qualname); // -> 3
} else if (co->co_flags & CO_ASYNC_GENERATOR) {
gen = PyAsyncGen_New(f, name, qualname);
} else {
gen = PyGen_NewWithQualName(f, name, qualname);
}
_PyObject_GC_TRACK(f);
return gen;
}
Objects/genobject.c (line no 1152)
PyObject * PyCoro_New(PyFrameObject *f, PyObject *name,
PyObject *qualname)
{
PyObject *coro = gen_new_with_qualname(&PyCoro_Type, f, name,
qualname); // -> 4
...
return coro;
}
```

Code insights are as

The call function internally calls the function **_PyEval_EvalCodeWithName** to create an instance of the coroutine. The tracing of this has been covered in the *Chapter_5_on_Functions and*

The compiler creates a flag called **CO_COROUTINE** during compilation to indicate that the created function is a coroutine.

The function **PyCoro_New** is used to create an instance of coroutine.

The function internally creates a generator of type **PyCoro_Type** to indicate that the object is a coroutine.

The following code block explains the structure of a coroutine:

Objects/genobject.c (line no 1152)
typedef struct {
_PyGenObject_HEAD(cr) // -> 1
PyObject *cr_origin;
} PyCoroObject;

A coroutine Python is basically implemented as a generator object, which contains a reference to the function being executed.

The following code block explains the implementation of the **YIELD_FROM** opcode:

```
case TARGET(YIELD_FROM): {
PyObject *v = POP();
PyObject *receiver = TOP();
int err;
if (PyGen_CheckExact(receiver) || PyCoro_CheckExact(receiver)) {
retval = _PyGen_Send((PyGenObject *)receiver, v); // -> 1
...
goto exit_yielding;
}
```

```
static PyObject* gen_send_ex(PyGenObject *gen, PyObject *arg, int
exc, int closing)
{
PyThreadState *tstate = _PyThreadState_GET();
PyFrameObject *f = gen->gi_frame;
PyObject *result;

if (gen->gi_running) {          // -> 2

const char *msg = "generator already executing";
if (PyCoro_CheckExact(gen)) {// -> 3
msg = "coroutine already executing";
}
else if (PyAsyncGen_CheckExact(gen)) {
msg = "async generator already executing";
}
PyErr_SetString(PyExc_ValueError, msg);
return NULL; Click here to enter text.
```

```
}
...
}
```

Code insights are as

Mark the paused generator to be running again.

Continue the execution of the code at the point left over previously very similar to other generators with the value provided to the send function.

Mark the running generator to be returned to the paused state again.

Continuing the execution of the coroutine

In the previous section, we covered how we could exit from the execution of a coroutine using the **YIELD_FROM** keyword. This section covers how the send function is used to resume the execution using the value provided to the **send** function:

Objects/genobject.c (line no 152)

```
static PyObject* gen_send_ex(PyGenObject *gen, PyObject *arg, int
exc, int closing)
{
PyThreadState *tstate = _PyThreadState_GET();
PyFrameObject *f = gen->gi_frame;
PyObject *result;
....
/* Generators always return to their most recent caller, not
* necessarily their creator. */
Py_XINCREF(tstate->frame);
assert(f->f_back == NULL);
f->f_back = tstate->frame;
gen->gi_running = 1; // -> 1
gen->gi_exc_state.previous_item = tstate->exc_info;
tstate->exc_info = &gen->gi_exc_state;
result = PyEval_EvalFrameEx(f, exc); // -> 2
tstate->exc_info = gen->gi_exc_state.previous_item;
gen->gi_exc_state.previous_item = NULL;
gen->gi_running = 0; // -> 3
```

...
}

Code insights are as

Mark the paused generator to be running again.

Continue the execution of the code at the point left over previously, very similar to other generators with the value provided to the **send** function.

Mark the running generator to be returned to the paused state again.

Asynchronous functions

Asynchronous Python functions are similar in nature to coroutines, where the interruption of the function can be paused in between and can be returned back using a callback function to which the generator is passed as an argument to continue the execution at the point where the **async** function was paused. This section will cover the basics of the workings of the **async** function.

Structure of the asynchronous generators:

```
typedef struct {

_PyGenObject_HEAD(ag) // -> 1
PyObject *ag_finalizer;

/* Flag is set to 1 when hooks set up by sys.set_asyncgen_hooks
were called on the generator, to avoid calling them more
than once. */
int ag_hooks_inited;

/* Flag is set to 1 when aclose() is called for the first time, or
when a StopAsyncIteration exception is raised. */
int ag_closed; // -> 2

int ag_running_async;
```

} PyAsyncGenObject;

Code insights are as

An asynchronous function is internally a generator, the execution of which is paused when the **await** statement is called. The same generator is added as a callback to the called function to return back to the same point post the completion of the awaited function.

The flag to indicate if the asynchronous function is successful/not.

Example of an asynchronous function to demonstrate the functionality of yielding from an **async** function:

```
import asyncio

async def sleeper():
await asyncio.sleep(10)
```

Disassembly of object sleeper at 0x10b0027b0, file "async.py", line 4>:

```
5           0 LOAD_GLOBAL              0 (asyncio)
2 LOAD_METHOD                1 (sleep)
4 LOAD_CONST                 1 (10)
6 CALL_METHOD                1
8 GET_AWAITABLE                // -> 1
10 LOAD_CONST                 0 (None)
12 YIELD_FROM                  // -> 2
```

```
14 POP_TOP
16 LOAD_CONST               0 (None)
18 RETURN_VALUE
```

Opcode insights are as

Fetch the generator of the asynchronous function.

Yield from the executing function to stop the execution of opcodes at the paused opcode.

Conclusion

This chapter covered the basics of coroutines and their implementation using generators. Async/await in Python are also implemented using generators, which are the fundamental building blocks of async programming in Python.

CHAPTER 10

Source Code Layout and the Compiler Stages

Understanding the fundamentals of a programming language helps build the basics upon which the peripherals can be explained. Programming languages are composed of grammar that defines the structure of the code and use compilers to translate the program into opcodes.

In programming languages such as C/C++, the compiler directly converts the program into machine-level opcode while compilers for languages such as Python convert it into opcode understood by the virtual machine/interpreter. This chapter covers the basics of Python grammar along with the parser, compiler, assembler, and opcode.

Structure

In this chapter, we will cover the following topics:

The folder structure of the Python source code

The main function

Python grammar

Stages of parsing and compilation

Operations on the parse tree

Navigation and conversion of the parse tree

Symbol table generation

Compilation of the AST to opcode

Objective

After reading this chapter, you will be able to understand the basics of Python processes of conversion of source code to opcode until the process of interpretation. You will also understand the code flow until the point of interpretation.

The folder structure of the Python source code

Going through the source of Python can be simplified by understanding the folder structure and reasoning of the structure. This section briefly covers each folder and its importance in the source tree:

Contains the documentation of the source code tree. The entire documentation is compiled as a list of **.rst** files, which can be converted to The converted **html** is hosted at the following site for access by the developer community. The folder also contains the copyright and license rights of the source code:

https://docs.python.org/3/

The folder contains the textual representation of the Python grammar and tokens as a **Backus Naur** form. The folder contains two files, file 1 named which defines the Python grammar, and file 2 named **Tokens,** which defines the tokens of the Python grammar.

Contains the libraries common to the entire folder structure that can be included in other parts of the source code. The libraries include the interfaces and storage structures for data structures, libraries, interpreter declarations, and exception management. These interfaces are also exposed as a part of the Python C API, which

can be used in the development of wrappers for API's developed in other programming languages.

The folder contains the Python implementation of modules that are part of the standard library. Some of the modules contain a C implementation for improved performance implemented in the **Modules** folder.

Contains the implementation of the core functionalities of Python, such as data structures, memory management, and the management of weak references, to mention a few.

Contains the implementation of the main function in Python that acts as the entry point of execution.

Contains the implementation of the core modules of Python that are the interpreter, error management, initialization of the compiler, optimization, and lifecycle management.

Contains the C implementation of the modules part of the standard Python library.

The main function

The **main** function begins to start the execution of the Python interpreter. The **main** function accepts the arguments to start the execution of the code from parsing, compiling, and interpreting the source code. It also loads the standard Python modules, initiates the data types, and creates the base thread to begin the interpretation of opcode:

Include/object.h Line no 104

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

#include "Python.h"
#include "pycore_pylifecycle.h"

...
int main(int argc, char **argv)
{
return Py_BytesMain(argc, argv); // -> 1
}
```

Code insight is as

The **main** function is kept minimal, which makes a call to **Py_BytesMain** function and performs the initialization of the interpreter.

The Python grammar

The textual representation of the grammar is present in the file It is written in **yacc** and is in the **Backus Naur form,** and is converted into a **non-definite finite automata** which is in the file

Grammar/Grammar
NOTE WELL: You should also follow all the steps listed at
https://devguide.python.org/grammar/

Start symbols for the grammar:
single_input is a single interactive statement;
file_input is a module or sequence of commands read from an input file;
eval_input is the input for the eval() functions.
func_type_input is a PEP 484 Python 2 function type comment
NB: compound_stmt in single_input is followed by extra NEWLINE!
NB: due to the way TYPE_COMMENT is tokenized it will always be followed by a NEWLINE
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER

decorator: '@' dotted_name ['(' [arglist] ')'] NEWLINE // -> 1
decorators: decorator+

decorated: decorators (classdef | funcdef | async_funcdef)

async_funcdef: ASYNC funcdef // -> 2
funcdef: 'def' NAME parameters ['->' test] ':' [TYPE_COMMENT]
func_body_suite // -> 3

...

Code insights are as

The grammar for **decorator** begins with followed by the name of the function and the open parentheses (and the list of arguments and the closing parentheses), followed by a new line.

The **async** function contains the keyword **async** followed by the definition/grammar of the function.

The definition of the function includes the **def** keyword followed by the name of the function and the input parameters.

The output of the Python grammar is the parse tree that contains the source file compiled as a tree of parsed nodes.

Parse tree to abstract syntax tree

The parse tree is converted into the **abstract syntax tree** by the AST generator, which is fed as input to the compiler. This section covers the conversion from the parse tree into the **abstract syntax tree**

Operations on the parse tree

Before understanding the implementation of the conversion of the parse tree to the AST, the implementation of few node traversal macros is explained as follows:

CHILD(node *, int) - Returns the nth child of the node using zero-offset indexing.

RCHILD(node *, int) - Returns the nth child of the node from the right side. Index is negative number.

NCH(node *) - Number of children of the node.

STR(node *) - String representation of the node; e.g., will return : for a COLON token

TYPE(node *) - The type of node as specified in Include/graminit.

REQ(node *, TYPE) - Assert that the node is the type that is expected

LINENO(node *) - retrieve the line number of the source code that led to the creation of the parse rule;

Navigation and conversion of the parse tree

Navigating the parse tree involves using the operations on the parse tree covered in the previous section and converting the same into the AST. The function, **PyAST_FromNodeObject** accepts the root of the parse tree and converts the same into an AST, which is explained as follows:

Python/ast.c (line no 793)

```
mod_ty PyAST_FromNodeObject(const node *n, PyCompilerFlags
*flags,
PyObject *filename, PyArena *arena)
{
mod_ty res = NULL;

...
switch (TYPE(n)) {
case file_input: // -> 1
stmts = _Py_asdl_seq_new(num_stmts(n), arena); // - > 2

if (!stmts)
goto out;

for (i = 0; i < NCH(n) - 1; i++) {// -> 3
ch = CHILD(n, i); // -> 4
if (TYPE(ch) == NEWLINE)
```

```
continue;
REQ(ch, stmt); // -> 5
num = num_stmts(ch); // -> 6
if (num == 1) {
s = ast_for_stmt(&c, ch) // ->
if (!s)
goto out; // -> 8
asdl_seq_SET(stmts, k++, s);


}
else {
ch = CHILD(ch, 0);
REQ(ch, simple_stmt);
for (j = 0; j < num; j++) {
s = ast_for_stmt(&c, CHILD(ch, j * 2)); // -> 9
if (!s)
goto out;
asdl_seq_SET(stmts, k++, s);
}
}
}
...
...
out:
if (c.c_normalize) {
Py_DECREF(c.c_normalize);
}
return res; // -> 10
}
```

Code insights are as

The inputs to the AST generator can be of type file, which is code input from a **.py** file, eval, or code input from the Python REPL interpreter or a module. This section covers the **file_input,** which encompasses all the other types.

Creating a new set of sequences to hold all the AST nodes for the current parse node.

Iterate through all the nodes in the current AST node.

Fetch the child node of the current AST node.

Check if the current node is mandatory/required.

Fetch the number of statements in the current AST node.

Fetch the AST representation of the current node.

If the created node is invalid, it raises a parse error.

When the number of statements in the node is *greater than* fetch the AST representation of all the nodes.

Return the created AST.

Symbol table generation

The symbol table is a data structure maintained by the Python compiler to store the meta information of compiled objects such as the variable, function names, classes, or objects. It is useful at different stages of program execution, such as compilation, execution, and optimization, to act as a store-specific for information or data.

The structure of the symbol table is as follows:

```
struct _symtable_entry;
struct symtable {
PyObject *st_filename; /* name of file being compiled, decoded from the filesystem encoding */ // -> 1
struct _symtable_entry *st_cur; /* current symbol table entry */ // -> 2
...
};
```

```
typedef struct _symtable_entry {
PyObject_HEAD
PyObject *ste_id; /* int: key in ste_table->st_blocks */ // -> 3
PyObject *ste_symbols; /* dict: variable names to flags */ // -> 4
PyObject *ste_name; /* string: name of current block */
PyObject *ste_varnames; /* list of function parameters */ // -> 5
...
```

int ste_lineno; /* first line of block */ // -> 6
struct symtable *ste_table; // -> 7
} PySTEntryObject;

Code insights are as

The filename of the file being compiled.

The current symbol table of the node being parsed example (class, function, or block).

The identifier of the entry is in the symbol table dictionary.

The symbols of the variables in the defined blocks along with the flags.

Function parameters if the symbol table corresponds to a function.

The **lineno** is in the code of to which the symbol table belongs.

The symbol table is associated with the entry.

Compilation to opcode

Compilation of the AST to opcodes is performed by the Python compilation module, which is then fed into the interpreter for execution. This section covers the conversion of the AST tree to opcodes. The **compiler_body** function accepts the AST tree as input and converts it into a set of opcodes:

Python/compile.c (line no 1746)

```
static int compiler_body(struct compiler *c, asdl_seq *stmts)
{
int i = 0;
stmt_ty st;
PyObject *docstring;
...
for (; i < asdl_seq_LEN(stmts); i++)
VISIT(c, stmt, (stmt_ty)asdl_seq_GET(stmts, i)); // -> 1
return 1;
}
```

Code insight is as

Visit all the elements in the AST tree and convert them into opcode one after the other.

This section will not cover the execution of the compiler in detail and is left to the readers to explore the compilation process in detail.

Conclusion

The compilation process forms the crucial part of understanding the internal workings of the Python programming language. Covering it in detail can itself be an entire volume on Python.

This chapter provided a simple introduction to the stages of the Python compilation process, which converted the program to a parse tree using the provided grammar. The parse tree is converted into an **abstract syntax tree** at the AST stage, which is used for both opcode generation and symbol table creation.

The opcode created at the compilation stage is then passed into the interpreter to begin execution.

A

B

C

D

F

G

I

L

M

N

O

P

R

S

T

U