# CRACKING THE TOUGH PARTS IN PYTHON

## SQUEEZE THE COBRA OUT OF YOUR MIND

ABDUR- RAHMAAN JANHANGEER

# Cracking The Tough Parts In Python

by Abdur-Rahmaan Janhangeer

build: 0.11.0

# Chapter: Foreword

This is a free and OpenSource book on Python.

The is only one criteria to chapters in this book: They must start from scratch and dive really, really deep.

You can

- 👉 [ view compileralchemy.com ] or
- 👉 [ read online ] or
- 👉 [ contribute to the book ] or
- 👉 [ download the book ] or
- 👉 [ support by buying on leanpub ]

Feel free to contribute a section, propose rewrites, fix typos etc. If you have comments, mail them to `arj.python at gmail dot com`.

# Chapter: Contributors

👉 Abdur-RahmaanJ: Main content

# Chapter: Decorators

Decorators occur prefixed by `@`. `@app.route` in the example below is a decorator.

```
@app.route('/home')
def index_page():
    pass
```

## Properties of functions

In order to understand decorators fully, it's great to know how functions behave.

### 1) Functions can be nested

Functions can be defined within functions

```
def calc(a, b):

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

    return add(a, b) * 2

print(calc(2, 2))
```

Functions can be defined within functions multiple times

```python
def calc(a, b):

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

    def minus(a, b):
        return a - b

    return add(a, b) * 2

print(calc(2, 2))
# 8
```

Functions can be defined within functions at multiple levels

```python
def do_this():
    def calc(a, b):
        def add(a, b):
            return a+b

        return add(a, b) * 2
    return calc(1, 3)

print(do_this())
# 8
```

## 2) Functions can take functions as arguments

A normal function

```python
def print_these():
    print('----')
    print('....')
    print('----')
```

calling / executing it

```python
print_these()
```

The symbols `(` `)` call the function

We can implement a fuction to execute other functions. It actually calls the function we pass in as argument

```python
def execute(f):
    f()
```

Applying

```python
def print_these():
    print('----')
    print('....')
    print('----')

execute(print_these) # same as print_these()

# ----
# ....
# ----
```

we can also retrieve values

```
def name():
    return 'moris'


def view_value(v):
    print('the value is', v())


view_value(name)

# the value is moris
```

## 3) Functions can return functions

Returning a fuction: here we are returning the function `y`.

```
def x():
    def y():
        print(3)
    return y
```

Calling `x` is the same as returning `y`. To get the value of y, we must add the `()` sumbols.

```
x()()
# 3
```

To avoid this (bit ugly)

```
x()()
```

We do

```
func = x()
func()
```

Example of use

```
def welcome_message():
    def first_part():
        return '------'

    def last_part():
        return '******'

    def body():
        return 'welcome to our program'

    def main():
        print(first_part())
        print(body())
        print(last_part())

    return main

w = welcome_message()

w()
```

prints out

```
# ------
# welcome to our program
# ******
```

## 4) Functions can be reassigned names

We can change function names by reassignment

```
def add(x, y):
    return x + y

addition = add

print(addition(2, 3))
# 5
```

This still works even if we delete the original function

```
def add(x, y):
    return x + y

addition = add

del add

print(addition(2, 3))
# 5
```

# Getting arguments passed

Python allows us to retrieve arguments passed to a function inside the function body itself.

## positional

we can get all arguments passed using `*args`

```
def s(*args):
    return args

print(s(1, 2, 3))
print(s(1, 2))
print(s(1))

# (1, 2, 3)
# (1, 2)
# (1,)
```

but we can change `*args` to anything like `*canne`.

```
def s(*canne):
    return canne

print(s(1, 2, 3))
print(s(1, 2))
print(s(1))

# (1, 2, 3)
# (1, 2)
# (1,)
```

This can be useful in the case of

```
def add(*nums):
    return sum(nums)

print(add(1, 2, 3, 4, 5))
print(add(100, 400, 1000))

# 15
# 1500
```

## keyword

`**kwargs` allows us to get all keyword arguments passed

```
def s(**kwargs):
    return kwargs

print(s(name='me', age=5,
country='mauritius'))

# {'name': 'me', 'age': 5, 'country':
'mauritius'}
```

As with `*args` we can change the name `kwarg` to what we like, for example `**keyword_arguments`

```
def s(**keyword_arguments):
    return keyword_arguments

print(s(name='me', age=5,
country='mauritius'))

# {'name': 'me', 'age': 5, 'country':
'mauritius'}
```

## Mixing args and kwargs

We can use `*args` and `**kwargs` within the same function

```
def view_args(*args, **keyw_args):
    print(args)
    print(keyw_args)

view_args(1, 2, 3, name='me', town='pl')
# (1, 2, 3)
# {'name': 'me', 'town': 'pl'}
```

# In enters the skeleton

Let us take this piece of code where we pass a function to another.

```
def quote(text):
    return '<<{}>>'.format(text)

def indent(text):
    return '>    {}'.format(text)

print(indent(quote('abc')))

# >    <<abc>>
#
# quote('abc') '<<abc>>'
# indent(quote('abc')) '>    <<abc>>'
```

We can also write it as

```python
def quote(text):
    return '<<{}>>'.format(text)

def indent(q):
    def dummy(text):
        return '>    {}'.format(
            q(text)
        )
    return dummy

print(
    indent(quote)('i am here')
)
# >    <<i am here>>
```

It is equivalent to:

```python
def indent(q):
    def dummy(text):
        return '>    {}'.format(
            q(text)
        )
    return dummy

@indent
def quote(text):
    return '<<{}>>'.format(text)

print(quote('the sun is rising'))
```

Which is neater!

# Chaining decorators

Let’s say we want to get the following output.

```
# ----
# >    <<the sun is rising>>
# ----
```

We just add another function

```
def enclose(f):
    def dummy(text):
        return '----\n{}\n----'.format(
            f(text)
        )
    return dummy
```

and just call

```
@enclose
```

```
def enclose(f):
    def dummy(text):
        return '----\n{}\n----'.format(
            f(text)
        )
    return dummy

def indent(q):
    def dummy(text):
        return '>    {}'.format(
            q(text)
        )
    return dummy

@enclose
@indent
def quote(text):
    return '<<{}>>'.format(text)

print(quote('the sun is rising'))
```

which results in

```
----
>    <<texthere>>
----
```

# Adding arguments to decorators

We can also add arguments to decorators.

```
def awesome_f(dec_param):
    def awesome_f_decorator(f):
        # wrap here
        def awesome_f_wrapper(p):
            # dec_param f(p)
        return awesome_f_wrapper
    return awesome_f_decorator

@awesome_f('abcd')
def some_func():
    # ...
```

# Bulletproofing our decorators

The above explanations need some fixing so as to be used in the real world. First, we need to accept all arguments.

```
def indent(q):
    def dummy(*args, **kwargs):
        return '>    {}'.format(
            q(*args, **kwargs)
        )
    return dummy
```

Then, we need to preserve information passed.

```
from functools import wraps
# ...
def indent(q):
    @wraps(func)
    def dummy(*args, **kwargs):
        return '>    {}'.format(
            q(*args, **kwargs)
        )
    return dummy
```

## Example of deveryday decorators

Here are some decorators you can encounter in everyday use.

### @staticmethod

Let's take a simple class

```
import math

class Calcs:

    def add(self, x, y):
        return x + y

    def hypotenuse(self, x, y):
        return math.sqrt((x**2) + (y**2))

c = Calcs()
print(c.hypotenuse(3, 4))

# 5.0
```

The functions not are not related together

Adding `@staticmethod`.

```
import math

class Calcs:

    @staticmethod
    def add(x, y): # self removed
        return x + y

    def hypotenuse(self, x, y):
        return math.sqrt((x**2) + (y**2))

print(Calcs.add(1, 2)) # no need of
instantiation

# 3
```

Uses of `@staticmethod`

- 👉 isolate function
- 👉 group related functions under a name space
- 👉 visually telling purpose of function

# @classmethod

syntax

```
@classmethod
def method_name(var_holding_classs, argument
...
```

Demo

```python
class Person:
    country = 'MU'

    @classmethod
    def say_hi(cls, name): # access class 
attributes through cls.
        return 'hi ' + name + ' from ' + 
cls.country

print(Person.say_hi('doe'))

# hi doe from MU
```

## @property

Another way of customising getters, setters and deleters

```python
class Car:
    def __init__(self):
        self._wheel = None

    @property
    def wheel(self):
        return self._wheel

    @wheel.setter
    def wheel(self, number):
        self._wheel = number

    @wheel.getter
    def wheel(self):
        return self._wheel

    @wheel.deleter
    def wheel(self):
        del self._wheel
```

Use

```
nissan = Car()
nissan.wheel = 4
print(nissan.wheel)
#4
```

but if getter changed

```
    @wheel.getter
    def wheel(self):
        return self._wheel + 1
```

and printed

```
nissan = Car()
nissan.wheel = 4
print(nissan.wheel)
```

we'd get `5`

Same as `__set__`, `__get__` and `__del__`

# Yeah, they are all functions

`@property`, `@staticmethod`, `@classmethod` are all functions, in-built ones. can be used as `property()`, `staticmethod()` and `classmethod()`.

try help on them

```
print(help(property))
```

# Chapter: Customising Behaviours Through Dunder Methods

In Python every element is an object. Each object has

👉 An identity (never changes once created, address in memory)
👉 A type
👉 A value

Objects also possess attributes starting and ending with `__` called dunder methods, e.g. `__doc__`.

```
>>> class X:
...     '''
...     fwfwefewf
...     '''
...     def __init__(self):
...             self.y = 10
...
>>> x = X()
>>> dir(x)
['__class__', '__delattr__', '__dict__',
'__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__',
'__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'y']
>>> x = X()
```

`x.__dict__()` returns `{'y': 10}`.

In Python, behind magic behaviours of classes lie special dunder methods.

What follows are some important dunder methods which customise the way Python features work.

# Class representations

Prehaps the simplest dunder method is `__repr__`. It enables us to customise the representation of our class. Here is a normal class.

```
class Fruit:
    pass
```

When printing the class, it will give us a default representation, `<__main__.Fruit object at 0x7f025a484f50>`.

```
apple = Fruit()
print(apple)
```

If we want to be more friendly to our users, we can customise what happens when a representation is needed.

```
class Fruit:
    def __repr__(self):
        return '<<Fruit Class>>'
```

When print is called, we now get `'<<Fruit Class>>'`.

`__repr__` should not be confused with `__str__` which defines what the string representation of an object is.

# Simulating numbers from a class

`__add__` and `__radd__` are called when encounering the `+` operator.

Here is a class which handles adding to other numbers as well as adding to classes of itself.

```
class Number:
    def __init__(self, val):
        self.val = val
    def __add__(self, other):
        print('__add__ called')
        return self.val + other
    def __radd__(self, other):
        print('__radd__ called')
        return self.val + other

x = Number(1)
y = Number(2)

print(x + 1)
print(1 + x)
print(x + y + 1)

# __add__ called
# 2
# __radd__ called
# 2
# __add__ called
# __radd__ called
# 4
```

As seen above, `__radd__` is called when the class is on the right hand-side of the operation. The *emulating numeric types* section of the docs has the complete list of descriptors to implement to fully simulate a number. As a side note, `__i.*__` for example `__iadd__` is used to implement `+=`.

# Context Managers

We can open file in Python as follows:

```
with open('filename') as f:
    ...
```

The `with` keyword hints to a context manager. The code enclosed in the with block is executed within a context. To implement a context manager, we implement the `__enter__` and `__exit__` descriptors. Before executing the code, whatever is defined in `__enter__` is executed. After executing the code, whatever is in `__exit__` is executed. This can be useful for pre and post processing.

Here is a toy example which sends a mail before and after a file is processed.

```python
def send_mail(message):
    print(message)

def notify_process(filename, action):
    send_mail('Process for '+filename+':
'+action)

class ProcessFile:
    def __init__(self, filename):
        self.filename = filename

    def __enter__(self):
        notify_process(self.filename,
'enter')
        return 'value of p'

    def __exit__(self, exc_type, exc_value,
traceback): # args None if exited without
exception
        notify_process(self.filename,
'exit')

pfile = ProcessFile('fruits.txt')

with pfile as p:
    print(p)

# Process for fruits.txt: enter
# value of p
# Process for fruits.txt: exit
```

Asynchronous context managers have the `__aenter__` and `__aexit__` methods.

# Iterators

Lists implement the iterator protocol as well as generators, as we'll see in the next chapter.

Here is a class which behaves like a list. It implements the `__iter__` and `__next__` methods. `__next__` returns the next element while iterating

```
class FlexibleNum:
    def __init__(self, num):
        self.num = num
    def __iter__(self):
        return self
    def __next__(self):
        self.num += 1
        return self.num - 1


x = FlexibleNum(5)
```

It will just increment the number to infinity.

```
for n in x:
    print(n)
```

To break, we can manually check the indices

```
for i, _ in enumerate(x):
    print(_)
    if i == 3:
        break
# 5
# 6
# 7
# 8
```

However, we can also bake in the end number in the class itself. We stop iteration by raising `StopIteration`.

```
class FlexibleNum:
    def __init__(self, num, end_number):
        self.num = num
        self.end_number = end_number
    def __iter__(self):
        return self
    def __next__(self):
        self.num += 1
        to_return = self.num-1
        if to_return >= self.end_number:
            raise StopIteration
        return to_return

x = FlexibleNum(5, 10)

for n in x:
    print(n)

# 5
# 6
# 7
# 8
# 9
```

As a side note, we could have incremented the number after the return statement by using `yield` instead of `return`.

We will cover generators in the next chapter.

# How attribute access works

Let's define another class which inherits from X.

```
>>> class X:
...     '''
...     fwfwefewf
...     '''
...     def __init__(self):
...             self.y = 10
...
>>> class B(X):
...     pass
...
>>> b = B()
>>> b.y
10
```

When accessing an attribute for example `b.y`, Python uses the Method Resolution Order (MRO) to determine if b has a y.

```
>>> type(b).mro()
[<class '__main__.B'>, <class '__main__.X'>,
<class 'object'>]
```

Python looks up on the type rather than the instace for performance reasons.

If we were to define a custom function for fetching y, it will be in brief:

```
def get_attribute(object_, attribute):
    # safety and elegance net removed
    for base_class in type(object_).mro():
        return
base_class.__dict__["__getattribute__"](obj,
attr)
```

Here we saw that `__getattribute__` is a dunder method used to preform attribute lookup.

## Acting like a dictionary

```
class NumToWord:
    def __init__(self):
        self.data = {1: 'one', 2: 'two'}
    def __getitem__(self, item):
        return self.data[item]

num2word = NumToWord()
print(num2word[1])
```

# Misc.

Descriptors are objects which define `__get__`, `__set__`, or `__delete__`. Personally, i prefer `@property` to set and get attributes.

# Refs


👉 [1] https://snarky.ca/unravelling-attribute-access-in-python/

# Chapter: Generators

In Python, generators form part of the intermediate topics. Since it differs from conventional functions, beginners have to take sometimes to wrap their head around it. This article presents materials that will be useful both for beginners and advanced programmers. It attempts to give enough to understand generators in depth but don't cover all use cases.

## Why were Python generators introduced?

Before we present generators and it's syntax, it's important to know why in the first place were generators introduced. The yield keyword does not mean generators. One has to understand the concept behind. The original PEP introduced "the concept of generators to Python, as well as a new statement used in conjunction with them, the yield statement" [1].

The general use case of generators is as follows:

> When a producer function has a hard enough job that it requires maintaining state between values produced, [1]

And more explicitly

> provide a kind of function that can return an intermediate result (“the next value”) to its caller, but maintaining the function’s local state so that the function can be resumed again right where it left off. [1]

So we understand that a new kind of functions was needed that:

👉 return intermediate values
👉 save the state of functions

## How do Python Generators differ from normal functions?

Compared to normal functions, once you return from a function, you can go back to return more values. A normal function in contrast, once you return from it, there is no going back.

Normal function:

```
def x():
    print('abc')
    return
    print('def') # not reached
x()
```

In the above example, `def` will not be printed as the function exited before. Let’s examine a basic generator example:

```
def x():
  a = 0
  while 1:
    yield a

    a += 1

z = x()

print(z)
print(next(z))
print(next(z))
```

```
<generator object x at 0x01ACB760>
0
1
```

From the example above, once we called next, it returned a value. **The purpose of next is to go to the next yield statement**. When we called next the first time, it went to the next yield since the beginning which is when `a` was initially at 0.

The second call of next started executing `a += 1` and went to the beginning of the loop where it encountered a yield statement and returned `a` with the updated value.

By `a` being updated we see that even when the function was exited the first time, when the program went back into it, it continued on the previous state when `a` was 0. This accomplishes the two aims of being able to resume functions and saving the previous state.

To understand it better, here are some more names that were proposed instead of yield [1] but were eventually rejected:

- 👉 return 3 and continue
- 👉 return and continue 3
- 👉 return generating 3
- 👉 continue return 3

Guido gives a summary of generators [1]:

> In practice (how you think about them), generators are functions, but with the twist that they're resumable.

## Execution flow

The following snippet gives us an idea about the execution:

```
def x():
  print('started')
  while 1:
    print('before yield')
    yield
    print('after yield')


z = x()

next(z)
print('-- 2nd call')
next(z)
```

```
started
before yield
-- 2nd call
after yield
before yield
```

From it we confirm that the first call to next executes everything in the function until the first yield statement. We did not return any values but used yield purely to control the flow of execution in the same sense of return.

Yield needs not to be in infinite loops, you can use several at once in the same function body:

```
def x():
  print('start')
  yield
  print('after 1st yield')
  yield
  print('after 2nd yield')

z = x()
next(z)
next(z)
next(z)
```

```
start
after 1st yield
after 2nd yield
Traceback (most recent call last):
  File "lab.py", line 11, in <module>
    next(z)
StopIteration
```

In case you called next more than there is yield statements, generator functions raise the `StopIteration`. In case you want to auto-handle `StopIteration` until there are no more left, use … a for loop:

```
def x():
  print('start')
  yield
  print('after 1st yield')
  yield
  print('after 2nd yield')

z = x()
for _ in z:
  pass
```

In case you return a value, the loop variable will be equal to that value:

```
def x():
  print('start')
  yield 1
  print('after 1st yield')
  yield 2
  print('after 2nd yield')

z = x()
for _ in z:
  print(_)
```

```
start
1
after 1st yield
2
after 2nd yield
```

## Immediate usefulness

Since we saw that we can use yield with an infinite loop, this is extremely powerful. We can break infinity in steps. Consider this:

```
def odd_till(number):
  n = 1
  while n < number:
    yield n
    n += 2

for odd_num in odd_till(10):
  print(odd_num)
```

We yield one number and the function exits, the for loop calls it again. It yields one number and exits. And so on. It goes about it in micro steps. The operations completed in one cycle is is just an increment `n += 2` and a check `n < number`.

---

> odd_till(10) or odd_till(1000000000000000000000000000000000000000000
> don’t not cause memory errors

## next and for loops

Two things might puzzle you:

- 👉 why was next used?
- 👉 how can a function with 2 yields work when a for loop is used with it?

The answer lies in in the fact that generators implement the iterator protocols, the same one used by lists. Here is a class customised to act as a generator [7]:

```
# Using the generator pattern (an iterable)
class firstn(object):
    def __init__(self, n):
        self.n = n
        self.num = 0

    def __iter__(self):
        return self

    # Python 3 compatibility
    def __next__(self):
        return self.next()

    def next(self):
        if self.num < self.n:
            cur, self.num = self.num,
self.num+1
            return cur
        else:
            raise StopIteration()

sum_of_first_n = sum(firstn(1000000))
```

## Generators introduced for memory saving

Consider a list comprehension:

```
sum([x*x for x in range(10)])
```

A generator expression is much, much more efficicent [2]:

```
sum(x*x for x in range(10))
```

This was the second addition in the generator story.

## Generators for tasks

Lets modify our two functions with print

```
def odd_till(number):
  n = 1
  while n < number:
    print('odd_till {} currently:
{}'.format(number, n))
    yield n
    n += 2

def even_till(number):
  n = 0
  while n < number:
    print('even_till {} currently:
{}'.format(number, n))
    yield n
    n += 2
```

Lets have a class to run functions

```python
from collections import deque

class RunFunc:
  def __init__(self):
    self._queue = deque()

  def add_func(self, func):
    self._queue.append(func)

  def run(self):
    while self._queue:
      func = self._queue.popleft()
      try:
        next(func)
        self._queue.append(func)
      except StopIteration:
        pass
```

## usage

```python
func_runner = RunFunc()

func_runner.add_func(odd_till(5))
func_runner.add_func(even_till(4))
func_runner.add_func(odd_till(6))
func_runner.run()
```

---

## output

```
odd_till 5 currently: 1
even_till 4 currently: 0
odd_till 6 currently: 1
odd_till 5 currently: 3
even_till 4 currently: 2
odd_till 6 currently: 3
odd_till 6 currently: 5
```

If we rename the same thing we get a mini task scheduler [4]

```
from collections import deque

class TaskScheduler:
  def __init__(self):
    self._queue = deque()

  def add_task(self, task):
    self._queue.append(task)

  def run(self):
    while self._queue:
      task = self._queue.popleft()
      try:
        next(task)
        self._queue.append(task)
      except StopIteration:
        pass
```

usage

```
scheduler = TaskScheduler()

scheduler.add_task(odd_till(5))
scheduler.add_task(even_till(4))
scheduler.add_task(odd_till(6))
scheduler.run()
```

Just a point of note, why do we remove a task (popleft) and readd it (append)?

```
        task = self._queue.popleft()
        try:
          next(task)
          self._queue.append(task)
        except StopIteration:
          pass
```

That's because if it was finished (exception raised) well and good, it will go straight to the except block. Else the .append will get executed.

In other words if task terminated, don't add it back else add it back.

## The send method

Generators support a way of sending values to generators

```python
def times2():
    while True:
        val = yield
        yield val * 2

z = times2()


next(z)
print(z.send(1))
next(z)
print(z.send(2))
next(z)
print(z.send(3))
```

```
2
4
6
```

This was an important addition. This passage explains why was send introduced and why it’s important in asyncio [6]:

> Python's generator functions are almost coroutines – but not quite – in that they allow pausing execution to produce a value, but do not provide for values or exceptions to be passed in when execution resumes … However, if it were possible to pass values or exceptions into a generator at the point where it was suspended, a simple co-routine scheduler or trampoline function would let coroutines call each other without blocking – a tremendous boon for asynchronous applications. Such applications could then write co-routines to do non-blocking socket I/O by yielding control to an I/O scheduler until data has been sent or becomes available. Meanwhile, code that performs the I/O would simply do something like this: `data = (yield nonblocking_read(my_socket, nbytes))` in order to pause execution until the `nonblocking_read()` coroutine produced a value.

yield was fundamentally changed with the addition of send.

- 👉 Redefine yield to be an expression, rather than a statement. The current yield statement would become a yield expression whose value is thrown away. A yield expression's value is None whenever the generator is resumed by a normal next() call.
- 👉 Add a new send() method for generator-iterators, which resumes the generator and sends a value that becomes the result of the current yield-expression. The send() method returns the next value yielded by the generator, or raises

StopIteration if the generator exits without yielding another value.

send(None) can also be used instead of the first next()

## Deriving send

How do we derive send? A tricky question indeed. Here's a mini snippet showing how [4]

```
from collections import deque

class ActorScheduler:
  def __init__(self):
    self._actors = { } # Mapping of names to
actors
    self._msg_queue = deque() # Message
queue

  def new_actor(self, name, actor):
    '''
    Admit a newly started actor to the
scheduler and give it a name
    '''
    self._msg_queue.append((actor,None))
    self._actors[name] = actor

  def send(self, name, msg):
    '''
    Send a message to a named actor
    '''
    actor = self._actors.get(name)
    if actor:
      self._msg_queue.append((actor,msg))

  def run(self):
    '''
    Run as long as there are pending
messages.
    '''
    while self._msg_queue:
      actor, msg = self._msg_queue.popleft()
      try:
        actor.send(msg)
      except StopIteration:
        pass

# Example use
if __name__ == '__main__':
  def printer():
    while True:
      msg = yield
      print('Got:', msg)

  def counter(sched):
    while True:
    # Receive the current count
      n = yield
```

```
        if n == 0:
          break
        # Send to the printer task
        sched.send('printer', n)
        # Send the next count to the counter
task (recursive)
        sched.send('counter', n-1)


  sched = ActorScheduler()
  # Create the initial actors
  sched.new_actor('printer', printer())
  sched.new_actor('counter', counter(sched))
  # Send an initial message to the counter
to initiate
  sched.send('counter', 100)
  sched.run()
```

The above can be expanded with more areas like ready, ready to read, ready to write and writing the appropriate code to switch between the areas and … you have a concurrent app. This is the basics of an operating system [4]. Using `sched.send` allows to have a loop beyond the recursion limit of python. The recursion limit is `import sys; sys.getrecursionlimit()` usually 1000. try `sched.send('counter', 1001)`.

## What is yield from?

Consider the following code:

```
def gen_alph():
  for a in 'abc':
    yield a

def gen_nums():
  for n in '123':
    yield n


def gen_data():
  yield from gen_alph()
  yield from gen_nums()


for _ in gen_data():
  print(_)
```

```
a
b
c
1
2
3
```

It behaves exactly as if the alphabet and number loops with their respective yields was inside gen_data.

“yield from is to generators as calls are to functions” as Brett Cannon puts it [8]

## The last part

Generators have a close method, caught by a GeneratorExit exception:

```python
def gen_alph():
  try:
    for a in 'abc':
      yield a
  except GeneratorExit:
      print('Generator exited')

z = gen_alph()
next(z)
z.close()
```

```
Generator exited
```

They also have a throw method to catch errors:

```python
def gen_alph():
  try:
    for a in 'abc':
      yield a
  except GeneratorExit:
      print('Generator exited')
  except Exception:
    yield 'error occured'

z = gen_alph()
next(z)
print(z.throw(Exception))
```

```
error occured
```

## The limit of generators: Infinity and Beyond

If you really want the best of Python generators the internet can give you copied over and over by Python sites, see David Beazley's 3 parts series:

- 👉 Generator Tricks for Systems Programmers
- 👉 A Curious Course on Coroutines and Concurrency
- 👉 Generators: The Final Frontier

# Refs


- 👉 [1] https://www.python.org/dev/peps/pep-0255/
- 👉 [2] https://www.python.org/dev/peps/pep-0289/
- 👉 [3] https://dev.to/abdurrahmaanj/add-superpowers-to-your-python-lists-using-this-feature-24nf
- 👉 [4] Python Cookbook, David Beazley
- 👉 [5] https://docs.python.org/3/library/asyncio-task.html
- 👉 [6] https://www.python.org/dev/peps/pep-0342/
- 👉 [7] https://wiki.python.org/moin/Generators
- 👉 [8] Brett Cannon: Python 3.3: Trust Me, It's Better Than Python 2.7

# Chapter: Bytecodes

Traditionally, this is how Python's execution ensured. Python was an interpreter ingesting source strings and executing instructions.

```
-------     ---------     ---------------
| src | --> | parse | --> | interpreter |
-------     ---------     ---------------
```

Since sometimes Python started making use of a Virtual Machine (VM)

```
-------
| src |
-------
   |
   v
------------
| compiler |
------------
   |
   V
-------------------
| virtual machine |
-------------------
```

Though it might sound complicated, a Virtual Machine is just a program.

This is how typically compilation occurs.

```
[ parse tree]
    ↓
[ ast ]
    ↓
[ bytecode generation ]
    ↓
[ bytecode optimisation ]
    ↓
[ flow control graph ]
    ↓
[ code object generation ]
```

## Hands-on Bytecode

To execute a Python file, we feed the file name ending in `.py` to the Python interpreter.

```
$ python3.10 main.py
```

We have the same result if we feen in a `.pyc` file.

```
$ python3.10 __pycache__/main.cpython-
310.pyc
```

> If the bytecodes are not being generated, we can use `-m compileall`.It is also used for creating cached bytecode files when installing libraries

The rough steps to view the bytecode instructions from `.pyc` files is to read the files in binary mode, transform it into a code obhects then disassemble it using `dis.dis`.

```
import marshal
import sys
import dis

header_size = 8
if sys.version_info >= (3, 6):
    header_size = 12
if sys.version_info >= (3, 7):
    header_size = 16
with open("__pycache__/main.cpython-310.pyc", "rb") as f:
    metadata = f.read(header_size)
    code_obj = marshal.load(f)
    dis.dis(code_obj)
```

We then have this output

```
  1           0 LOAD_CONST               0 (1)
              2 STORE_NAME               0 (x)

  2           4 LOAD_CONST               1 (2)
...
```

Python provides the `compile()` function in-built.

```
>>> help(compile)
Help on built-in function compile in module
builtins:

compile(source, filename, mode, flags=0,
dont_inherit=False, optimize=-1, *,
_feature_version=-1)
    Compile source into a code object that
can be executed by exec() or eval().

    The source code may represent a Python
module, statement or expression.
    The filename will be used for run-time
error messages.
    The mode must be 'exec' to compile a
module, 'single' to compile a
    single (interactive) statement, or
'eval' to compile an expression.
    The flags argument, if present, controls
which future statements influence
    the compilation of the code.
    The dont_inherit argument, if true,
stops the compilation inheriting
    the effects of any future statements in
effect in the code calling
    compile; if absent or false these
statements do influence the compilation,
    in addition to any features explicitly
specified.
```

We can use it to transform source codes into code objects. Compile also serves to show that Python indeed has a compilation step occuring.

```
src = '''
x = 1
y = 2

print(x+y)
'''

c = compile(src, '', "exec")
exec(c)
# exec(src)
```

Here is what a code object is about

```
>>> help(c)
Help on code object:

class code(object)
 |  code(argcount, posonlyargcount,
kwonlyargcount,
 nlocals, stacksize, flags, codestring,
constants,
 names, varnames, filename, name,
firstlineno,
 linetable, freevars=(), cellvars=(), /)
 |
 |  Create a code object.  Not for the faint
of heart.
...
```

Bytecode instructions are ready to be executed using `exec`.

```
>>> help(exec)
Help on built-in function exec in module
builtins:

exec(source, globals=None, locals=None, /)
    Execute the given source in the context
of globals
    and locals.

    The source may be a string representing
one or more
    Python statements
    or a code object as returned by
compile().
    The globals must be a dictionary and
locals can be any
    mapping,
    defaulting to the current globals and
locals.
    If only globals is given, locals
defaults to it.
```

Code objects have interesting attributes. Attributes prefixed by `.co_` are of particular interest. The `.co_code` attribute holds the bytecode content in bytes.

```
>>> c.co_code
b'd\x00Z\x00d\x01Z\x01e\x02e\x00e
\x01\x17\x00\x83\x01\x01\x00d\x02S\x00'
>>> type(c.co_code)
<class 'bytes'>
```

Looping over it gives us the bytes in numbers. I formatted the output so as to have 2 elements on a line.

```
>>> [c for c in c.co_code]
[
100, 0,
90, 0,
100, 1,
90, 1,
101, 2,
101, 0,
101, 1,
23, 0,
131, 1,
1, 0,
100, 2,
83, 0
]
```

# Bytecodes are introduced

A bytecode instruction looks like this

```
LOAD_CONST 2
```

`LOAD_CONST` is the opcode while `2` is the oparg.

```
LOAD_CONST 2 op arg

  opcode
```

`dis.HAVE_ARGUMENT` is a number. Any bytecode represented by a number above it takes argument. `dis.HAVE_ARGUMENT` can be 30 for example. If i have a bytecode represented by 40, i know it will take arguments.

`dis.opname` is a dictionary for translating the opcode number into a string representation. Here is a snippet for listing opcodes number and name from a code object.

```
>>> import dis
>>> [(dis.opname[c] if i%2==0 else c)
        for i, c in enumerate(c.co_code)]
[
    'LOAD_CONST', 0,
    'STORE_NAME', 0,
    'LOAD_CONST', 1,
    'STORE_NAME', 1,
    'LOAD_NAME', 2,
    'LOAD_NAME', 0,
    'LOAD_NAME', 1,
    'BINARY_ADD', 0,
    'CALL_FUNCTION', 1,
    'POP_TOP', 0,
    'LOAD_CONST', 2,
    'RETURN_VALUE', 0
]
```

# Dissecting bytecodes

Typically, the code to be inspected is placed in a function. Then the function is disassembled using `dis.dis`.

```
>>> def func():
...     x = 1
...     y = 1
...     print(x+y)
...
>>> dis.dis(func)
  2           0 LOAD_CONST               1
(1)
              2 STORE_FAST               0
(x)

  3           4 LOAD_CONST               1
(1)
              6 STORE_FAST               1
(y)

  4           8 LOAD_GLOBAL              0
(print)
             10 LOAD_FAST                0
(x)
             12 LOAD_FAST                1
(y)
             14 BINARY_ADD
             16 CALL_FUNCTION            1
             18 POP_TOP
             20 LOAD_CONST               0
(None)
             22 RETURN_VALUE
```

In the above example, `2`, `3`, `4` are line numbers. `0`, `2`, `4`, `6` are the opcode index. It is a number assigned to the opcode line numeber. It is used for jumps.

`.co_names`, `.co_varnames` and `.co_consts` hold the remaining values.

```
>>> func.__code__.co_names
('print',)
>>> func.__code__.co_varnames
('x', 'y')
>>> func.__code__.co_consts
(None, 1)
```

Free variables are variables used in a code block but not defined there. It is not applied to global vars.

# Associated concepts

When inspecting a piece of code, we return an array of frames.

```
inspect.stack() -> [
    FrameInfo(frame, filename, lineno,
    function, code_context, index), ...]
```

Values and results live on stacks.

`BINARY_ADD` works by poping. two values from the stack, operates on them then places the result back.

`cpython/Include/opcode.h` has a list of opcodes. There are some 191.

Frames contain contextual info about stack and interpreter states. They are attached to a thread.

Each module, function and class has a frame [2]

Generators switch frames, but need a data stack for each frame

There is a frame for each code object

A stack of frames possible (call stack).

`RETURN_VALUE` instructs to pass value between frames

There are 2 stacks: Call and data stack.

# How are bytecodes executed

`cpython/Programs/python.c` has main (or wmain)

It calls `Py_BytesMain` or `Py_Main` from `modules/main.c`, both calling same thing with different args.

The bytecode execution itself is enclosed in a big switch statement.

```
switch (opcode) {
    // ...
case TARGET(BINARY_ADD): {
            PyObject *right = POP();
            PyObject *left = TOP();
            PyObject *sum;
            /* NOTE(haypo): Please don't try
to micro-optimize int+int on
               CPython using bytecode, it is
simply worthless.
               See
http://bugs.python.org/issue21955 and

http://bugs.python.org/issue10044 for the
discussion. In short,
               no patch shown any impact on
a realistic benchmark, only a minor
               speedup on microbenchmarks.
*/
            if (PyUnicode_CheckExact(left)
&&

PyUnicode_CheckExact(right)) {
                sum =
unicode_concatenate(tstate, left, right, f,
next_instr);
                /* unicode_concatenate
consumed the ref to left */
            }
            else {
                sum = PyNumber_Add(left,
right);
                Py_DECREF(left);
            }
            Py_DECREF(right);
            SET_TOP(sum);
            if (sum == NULL)
                goto error;
            DISPATCH();
        }
```

Bytecodes not same for all versions of Python.

# Working of common opcodes

`BINARY_ADD` retrieves values from the stack, operates on them then places the result back.

```
BINARY_ADD

[1, 2]
 ⬇
[]
 ⬇
[3]
```

```
LOAD_CONST

[]
 ⬇
[5]
```

```
STORE_FAST

[5]
 ⬇
[]
```

# Assignment

```
x = 1
```

```
  1           0 LOAD_CONST               1
(1)
              2 STORE_FAST               0
(x)
```

## Conditionals

```
if x < 2:
    return True
```

```
  2           0 LOAD_CONST               1
(1)
              2 LOAD_CONST               2
(2)
              4 COMPARE_OP               0
(<)
              6 POP_JUMP_IF_FALSE        6
(to 12)

  3           8 LOAD_CONST               3
(True)
             10 RETURN_VALUE

  2     >>   12 LOAD_CONST               0
(None)
             14 RETURN_VALUE
```

## While loops

```
x = 10
while x < 20:
        x += 2
```

```
  2           0 LOAD_CONST               1
(10)
              2 STORE_FAST               0
(x)

  3           4 LOAD_FAST                0
(x)
              6 LOAD_CONST               2
(20)
              8 COMPARE_OP               0
(<)
             10 POP_JUMP_IF_FALSE       16
(to 32)

  4     >>   12 LOAD_FAST                0
(x)
             14 LOAD_CONST               3
(2)
             16 INPLACE_ADD
             18 STORE_FAST               0
(x)

  3          20 LOAD_FAST                0
(x)
             22 LOAD_CONST               2
(20)
             24 COMPARE_OP               0
(<)
             26 POP_JUMP_IF_TRUE         6
(to 12)
             28 LOAD_CONST               0
(None)
             30 RETURN_VALUE
        >>   32 LOAD_CONST               0
(None)
             34 RETURN_VALUE
```

# The Question of Platform

The VM is not a platform

Compiled codes may break for the next version as the developers reserve the right to change opcode operations.

Currently

```
Python VM
[ stuffs ] -> [ bytecode ] -> [ optimised
bytecodes ]

SQLite VM
[ stuffs ] -> [ optimise ] -> [ bytecode ]
```

Since optimisations are done after the bytecode has been fed, this opens the door for the VM to be used as a platform. Maybe in the future we can have people writing bytecodes and targetting the VM. Think Kotlin/Java.

# Dissy: A TUI disaasmbler

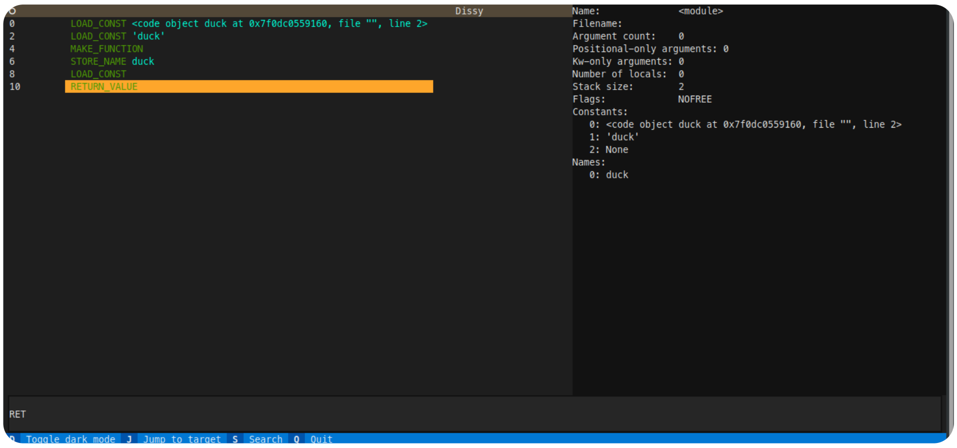
Dissy allows us

```python
src = '''
def duck():
    x = 1
'''

c = compile(src, '', "exec")

import dissy
dissy.dis(c)
```

```
python -m pip install dissy click distorm3
```




# Interesting Bits

## 1) Quote from a C file.

> Function objects and code objects should not be confused with each other:
>
> Function objects are created by the execution of the 'def' statement. They reference a code object in their `__code__` attribute, which is a purely syntactic object, i.e. nothing more than a compiled version of some source code lines. There is one code object per source code "fragment", but each code object can be referenced by zero or many function objects depending only on how many times the 'def' statement in the source was executed so far. [4]

2) PEP617 - Python3.9 uses a PEG-based parser (PEG - 2004)

Though the parser was top-down, it does not respect the rules of top-down and workarounds were used. Also, the Intermediate Represetation (parse tree or Concrete Syntax Tree) was around just for the sake of it.

# Refs


- 👉 [1] Inside The Python VM, Obi Ike-Nwosu
- 👉 [2] A Python Interpreter Written in Python, Allison Kaptur, Ned Batchelder
- 👉 [3] Understanding Python Bytecode, Reza Bagheri https://www.linkedin.com/in/reza-bagheri-71882a76/
- 👉 [4] https://github.com/python/cpython/blob/3db0a21f731cec28a89f7495a82ee2670b
- 👉 [5] https://tenthousandmeters.com/blog