Python

Muchang Bahng

Fall 2024

Contents

After coding in Python for about 4 to 5 years, I realized that my coding practices have not changed, and I should try to grow on them. These notes have four purposes. Learn some intermediate Python through different syntax, methods, and classes. Learn how Python and its data structures are implemented, specifically CPython (C notes are previously done). Establish best practices by going through different case studies of codebase design. Learn the APIs of some broad Python packages, mostly in the standard library that are pretty up in the dependency tree.

All of these can be found in either:

1. The official Python language reference, which describes the exact syntax and semantics of the Python language.
2. The official Python standard library, which describes the standard library (the built-in modules) that is distributed with Python.
3. The Index of Python enhacement proposals (PEP), which is a series of design documents providing information to the Python community. It is used to describe new features of Python and its processes of development.

Definition 0.1 (Object)

Every object has an **identity**, a **type**, and a **value**.

Theorem 0.1 ()

In CPython, `id(x)` is the memory address where `x` is stored.

1 Lexical Analysis

When we have code in a `.py` file and run it, the **lexical analyzer** generates a stream of **tokens** to be inputted into a parser.

Theorem 1.1 ()

All UTF-8 characters can be parsed by the lexical analyzer.

Question 1.1 ()

Which characters aren't?

Definition 1.1 (Logical and Physical Lines)

There are two types of lines in Python.

1. A **logical line** is represented by the token `NEWLINE`.
2. A **physical line** is a sequence of characters terminated by an end-of-line (EOL) sequence.

```python
x = [1, 2, 3, 4]  # one logical line on one physical line
x = [1, 2,        # one logical line on two physical lines
     3, 4]
```

2 Types

Question 2.1 (To Do)

Move some of these to general language notes.

To avoid confusion, let's delve into a bit of history. During the early days of Python 2, the language had both *types* and *classes*. Types were built-in objects implemented in C, and classes were what you built when using a `class` statement. These two were named differently because you couldn't mix these; classes could not extend types. However, this difference was artificial and ultimately a limitation in the language implementation. Starting with Python 2.2, the developers of Python have slowly moved towards unifying the two concepts, which the difference completely done in Python 3. Built-in types are now labeled classes, and you can extend them at will. Since we are working in Python 3, they are interchangeable.

Theorem 2.1 (Types and Classes)

In Python 3, types and classes mean the same thing.

Definition 2.1 (Type Checking)

Type checking is the process of verifying that the types of values in a program are used consistently and correctly according to the language's type system rules. These include:

1. Operations are valid for their operand types
2. Function/method calls match their signatures
3. Assignments are type-compatible (though this isn't necessary in Python)

The implementation of type checking differs for every language, and they generally fall into 3 different philosophies.

Definition 2.2 (Nominal Typing)

Nominal typing is a static typing system that determines that two types are equal/compatible if their fully qualified class names (FQCN) are equal.

```cpp
struct Cat {
    std::string name;
    int age;
};

void printCat(const Cat& c) {
    std::cout << "Cat: " << c.name << ", age " << c.age << "\n";
}
```

```cpp
struct Dog {
    std::string name;
    int age;
};

void printDog(const Dog& d) {
    std::cout << "Dog: " << d.name << ", age " << d.age << "\n";
}
```

```cpp
int main() {
    Cat kitty{"Whiskers", 3};
    Dog pup{"Buddy", 5};

    printCat(kitty); // works
    printDog(pup);   // works

    // printCat(pup); // error: cannot convert Dog to Cat (nominal typing)

    return 0;
}
```

Figure 1: C++ uses aspects of nominal typing.

Definition 2.3 (Structural Typing)

Structural typing is a static typing system that determines that two types are equal/compatible if their structures (e.g. the attributes and methods it supports) are equal. The class name is immaterial.

```typescript
type Cat = {
  name: string;
  age: number;
};

function printCat(c: Cat) {
  console.log(`Cat: ${c.name}, age ${c.age}`);
}
```

```typescript
type Dog = {
  name: string;
  age: number;
};

function printDog(d: Dog) {
  console.log(`Dog: ${d.name}, age ${d.age}`);
}
```

```typescript
const kitty: Cat = { name: "Whiskers", age: 3 };
const pup: Dog = { name: "Buddy", age: 5 };

printCat(kitty); // works
printDog(pup);   // works
printCat(pup);   // also works (structural typing!)
```

Figure 2: Typescript uses aspects of structural typing.

Definition 2.4 (Duck Typing)

Duck typing is a dynamic typing system that determines that two types are equal/compatible is the *accessed* structure (e.g. used attributes or called methods) are equal. The class name and the unused properties are immaterial.[a]

```python
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def meow(self):
        print("meow")

def print_cat(c):
    print(f"Cat: {c.name}, age {c.age}")
```

```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("woof")

def print_dog(d):
    print(f"Dog: {d.name}, age {d.age}")
```

```python
kitty = Cat("Whiskers", 3)
pup   = Dog("Buddy", 5)

print_cat(kitty) # works
print_dog(pup)   # works
print_cat(pup)   # also works though structures are different
pup.bark(), kitty.meow()       # works
pup.meow() # Error: 'Dog' object has no attribute 'meow'
```

Figure 3: Python uses duck typing: any object with the right attributes can be passed.

Duck typing and structural typing are similar (and often confused) but distinct, and the preference for one over the other is controversial. The big difference is that duck typing is "looser" in that type checking happens at *runtime*, whether an object has the required methods/properties when they are actually used.

We will start by going through all the types in Python.

[a]If it walks like a duck and quacks like a duck, then it must be a duck.

3 Primitives

3.1 String Manipulation

Definition 3.1 (Checking Alphanumeric)

Method	
`str.isalnum()`	Return `True` is all chars in are alphanumeric and there is at least 1 char.
`str.isalpha()`	Return `True` if all characters in string are alphanumeric and there is at least 1 char.

Table 1

You probably used the `str.strip()` method. However, you can have more control over this.

Definition 3.2 (Strip, Prefix, and Suffix)

Method	
`str.lstrip(chars=None)`	Returns copy of string with leading characters (default ascii space) removed.
`str.rstrip(chars=None)`	Returns copy of string with trailing characters (default ascii space) removed.
`str.strip(chars=None)`	Returns copy of string with both leading and trailing characters removed.
`str.removeprefix(prefix)`	Returns a string with the prefix removed (if it exists).
`str.removesuffix(suffix)`	Returns a string with the suffix removed (if it exists).
`str.startswith(prefix)`	Return `True` if starts with `prefix`, else `False`
`str.endswith(prefix)`	Return `True` if ends with `prefix`, else `False`

Table 2: Note that stripping, which targets all combinations defined in `chars`, is more aggressive than removing prefix.

Definition 3.3 (Justify and Filling)

Method	
`str.ljust(width, fillchar=' ')`	Returns the string left justified in a string of length width with padding `fillchar`.
`str.rjust(width, fillchar=' ')`	Returns the string right justified in a string of length width with padding `fillchar`.
`str.zfill(width)`	Returns copy of string left-filled with `"0"` digits to make a string of length `width`. Accounts for negative numbers.

Table 3: Note that stripping, which targets all combinations defined in `chars`, is more aggressive than removing prefix.

```
>>> "hello world".ljust(20)
'hello world         '
```

```
>>> "hello world".rjust(20)
'         hello world'
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

Definition 3.4 (Find, Index, and Replace)

Method	
`str.find(sub)`	Return the lowest index in string where substring `sub` is found. Returns -1 if not found.
`str.index(sub)`	Like `str.find(sub)`, but raises `ValueError` when substring is not found.
`str.replace(old, new)`	Return a copy of string with all occurrences of substring `old` replaced by `new`.
`str.translate()`	Replace all occurrences of characters in string with a translation table.

Definition 3.5 (Split and Partition)

Method	
`str.split(sep=None)`	Return a list of words in the string, using `sep` as delimiter string.
`str.splitlines(sep=None)`	Like `str.split()` but we account for all newline characters (not only just `\n`).
`str.partition(sep)`	Split the string at the first occurrence of `sep`, and return a 3-tuple.
`str.rpartition(sep)`	Split the string at last occurrence of `sep`, return a 3-tuple.

3.2 Typecasting

Let's talk about typecasting between these primitives. Note that converting strings to ints is pretty ambiguous.

Function	Input	Output	Notes
`int.to_bytes()`	`int`	`bytes`	Specify the `length` arg to prevent overflow.
classmethod `int.from_bytes()`	`bytes`	`int`	
`str.encode()`	`str`	`bytes`	Usually we use `encoding='utf-8'`.
`byte.decode()`	`bytes`	`str`	Usually we use `'utf-8'`.
`str()`	`int`	`str`	

Now if you want to convert this to a fixed length, then you can simply use the built-in `hash()` function. Ints, strings, and bytes are all immutable and thus hashable.

4 Data Structure

4.1 Lists

Lists are implemented as an array of pointers, which can point to any object in memory which is why Python lists can be dynamically allocated. We should be familiar with the general operations we can do with a list, which are implemented as dunder methods.

Definition 4.1 (Length)

The `list.__len__()` method returns the length of a list, which is stored as metadata and is thus $O(1)$ retrieval time. It is invoked by `len(list) <-> list.__len__()`.

Definition 4.2 (Set Item, Get Item, Del Item)

The following three methods are getter, setter, and delete functions on the `list[T]` array given the index.

1. The `__getitem__(i) -> T` returns the value of the index of the list. Since we can do pointer arithmetic on the array, which is again just 8 byte pointers, we essentially have $O(1)$ retrieval time. It is invoked by `list[i] <-> list.__getitem__(i)`.
2. The `__setitem__(i, val) -> None` returns `None` and sets the value of the index. It is invoked by `list[i] = val <-> list.__setitem__(i, val)`.
3. The `__delitem__(i) -> None` deletes the value at that index. It is invoked by `del list[i] <-> list.__delitem__(i)`.

The next few definitions are not dunder methods, but are important.

Definition 4.3 (Append, Insert, Pop)

`List.append(val)` is amortized $O(1)$ but is quite slow if we are inserting into the middle with `List.insert(i, val)`. `List.pop()` is great for removing from the back of the list, with $O(1)$, but not so great for removing from the front, where all the elements have to be shifted $O(n)$. Dynamically resizing the array, where all the elements of the previous array gets copied over to a larger array, is slightly different. For example, in an old implementation of Python, the new size is implemented to be `new_size + new_size » 3 + (new_size < 9 ?  3 :  6)`, which approximately doubles the size (like Java, which exactly doubles the list size), giving us amortized $O(1)$.

Definition 4.4 (Extend)

Definition 4.5 (Sort)

List slicing is quite slow since we are copying the references to every element in the list. Note that the values are not copied themselves, but we are creating an array of new pointers.

Slicing can be done past last index. Slicing creates a copy of the sublist.

Definition 4.6 (Queues)

A `collections.deque` (double ended queue) is implemented as a doubly linked list.

4.2 Hash Maps

In general, a hashmap can be implemented in the following ways. We take an object and hash its *value*, giving us another memory address. This intuitively implies that this object is immutable, since changing the object will lead to a different memory address. A convenient way to bypass this is to convert lists into tuples.[1] The hash function may map two different values to the same memory address, so we can deal with collisions in different ways.[2]

1. *Linked List.* The hashed address actually is a linked list, and every time we add to it we append to the linked list.
2. *Probing.* If we have two objects x_1 and x_2 which both map to the same $y = h(x_1) = h(x_2)$, then we can predefine another function f that will act on $h(x_2)$ when it sees that $h(x_1)$ is already occupied, effectively mapping it to $f(h(x_2))$. Two common ones is $f(x) = x + 1$, which maps it to the next address, called *linear probing*, or we can scale it in different ways, e.g. *quadratic probing*.
3. *Double Hashing, Open Addressing.* We can hash the hash differently, effectively doing $(h_1(x) + i \cdot h_2(x)) \text{mod} S$, and keep incrementing i from 0 to whenever it sees a new spot.

Definition 4.7 (Python Dictionaries)

Python does indeed implement dictionaries as hash maps/tables and uses open addressing to handle collisions, meaning that it can only store one and only one entry. Python's hash table is also a contiguous block of memory, so you can actually do $O(1)$ lookup by index as well, though the indices aren't stored.

```
-+-----------------+
0| <hash|key|value>|
-+-----------------+
1|       ...       |
-+-----------------+
.|       ...       |
-+-----------------+
i|       ...       |
-+-----------------+
.|       ...       |
-+-----------------+
n|       ...       |
-+-----------------+
```

Figure 4: Logical model of Python Hash table. It consists of the keys, the hash of the keys, and the values that are stored in the hashed memory address. The indices are shown on the left, but they are not stored along with the table.

When a new dict is initialized, it starts with 8 slots.

1. When adding entries to the table, we take the key k, hash it to h, and we do an additional mask operation `i = mask(key) & mask`, where `mask = PyDictMINSIZE - 1` (in CPython).
2. If the slot is empty, the entry is added to the slot. If the slot is occupied, CPython (and PyPy) compares the hash and the key (with `==`, not `is`) of the entry in the slot against what we are inserting. If *both* match, it thinks the entry already exists and uses open addressing to move onto the next entry.
3. The dict will be resized if it is 2/3 full to avoid slowing down lookups.

[1] However, there are languages where you can hash mutable objects. Again, this is an implementation detail.
[2] Good visuals here: https://www.geeksforgeeks.org/open-addressing-collision-handling-technique-in-hashing/.

It is well known that the keys and hash tables are not guaranteed to be in sorted order, and this is true in general. However, in Python it is different.

Theorem 4.1 ()

From Python 3.7+ (for all implementations) and CPython 3.6+, dicts preserve insertion order, so calling `dict.keys()` will return keys in insertion order

Example 4.1 (Back to References)

As a review, when we iterate over a dict with an enhanced for loop, we are just calling `next` on the keys or values that may be a copy by value or a copy by reference.

```
# y is copied by value so incrementing
# it rebinds it
>>> x = {"a" : 1, "b" : 2, "c" : 3}
>>> for k in x:
...     y = x[k]
...     y += 1
...
>>> x
{'a': 1, 'b': 2, 'c': 3}
```

```
# v is passed by value, so incrementing
# it rebinds it
>>> x = {"a" : 1, "b" : 2, "c" : 3}
>>> for v in x.values():
...     v += 1
...
>>> x
{'a': 1, 'b': 2, 'c': 3}
.
```

We should also be familiar with some of the dunder methods.

Definition 4.8 (Get)

There are two ways to access from a dictionary.
1. `dict[key]` retrieves the value and throws a `KeyNotFoundError` if a key does not exist.
2. `dict.get(key, def)` retrieves the value and will return `def` if the key does not exist.

Definition 4.9 (Items)

Given a dictionary `dict`, we can run `dict.items()` to get a *view* of the dictionary. Since this is a view, it does not copy the entire dictionary, and is presented as a list of tuples. However, this is not an iterator either. T

Let's look through the different dict-like data structures.

Definition 4.10 (Defaultdict)

A nice trick is to initialize a `collections.defaultdict`, which is a subclass of `Dict` that allows you to use `dict[key]` and automatically initializes the value to some default value if the key does not exist. It is initialized in the following ways.
1. `defaultdict(int)`
2. `defaultdict(dict: Dict)`
3. `defaultdict(log: Function, dict)` runs the function `log` every time a new key is added.

Definition 4.11 (Counter)

`collections.Counter` is good for finding the count of elements and does not require you to initialize the count to 0 before incrementing it.

```python
data = [1, 1, 2, 3]
counter = {}
for d in data:
    if d not in counter:
        counter[d] = 0
    counter[d] += 1
{1: 2, 2: 1, 3: 1}
```

```python
from collections import Counter
data = [1, 1, 2, 3]
counter = Counter()
for d in data:
    counter[d] += 1
Counter({1: 2, 2: 1, 3: 1})
.
```

4.3 Heaps

5 Names and Values

There are a lot of parallel characteristics between python variable assignment and C++ pointers. When we assign a variable to an object in python, what we are doing under the hood is creating the value/object in the heap memory (hence we use `malloc` rather than initializing on the stack) and initializing a pointer to point to that place in memory.

The left hand side is called a **name**, or a **variable**, and the right hand side is called the **value**. We say *the name references, is assigned, or is bound to the value.* In fact, this name is really just a pointer to the memory location of where the value is stored, and we can access this using the built-in `id` function.

```python
# Python
x = 4
print(x) # 4
print(id(x)) # 4382741696
.
.
```

```c
# C
int* x_ = malloc(sizeof(int));
*x_ = 4;
int** x = &x_;
printf("%d\n", **x); // 4
printf("%p\n", *x);  // 0x600003ff4000
```

Figure 5: Referencing an int variable in Python and C. I realize that this isn't completely equivalent since the C code uses a pointer to a pointer, but it helps explain other things a bit easier so bear with me.

```python
# Python
y = [1, 2, 3]
print(y)        # [1, 2, 3]
print(id(y))    # 4314417472
.
.
.
.
```

```c
# C
int* x_ = malloc(sizeof(int) * 3);
x_[0] = 1; x_[1] = 2; x_[2] = 3;
int** x = &x_;
for (int i = 0; i < 3; ++i) {
  printf("%d ", *(*x+i)); // 1 2 3
}
printf("\n%p", *x);  // 0x6000011cc040
```

Figure 6: Referencing a list in Python and C.

5.1 Mutating vs Rebinding

So far so good. But what if we wanted to change `x` or `y`? This is where we have to be careful about when defining *change*.

1. We can change by taking the value that the name references/points to and *mutate* it. Types of values where we can do this are called *mutable types*, which have methods that allow this change (e.g. `__setitem__` or `append` for lists). In this case, the memory address it points to should stay the same.
2. We can change by creating a new value/object and changing the name to point to this new object. If no other variables points to the original object, then the memory is automatically freed. This is how *immutable types* are changed, and the memory address it points to should be different. What immutable really means is that you cannot change the value that the pointer is pointing to without changing the actual memory location.

So which one is it that Python does? The answer is: it depends.[3]

[3]For more information, look at https://nedbatchelder.com/text/names.html.

Example 5.1 (Pass By Reference vs By Value)

There are two ways a programmer can interpret the following iconic example.

```python
x = 4
y = x
print(x, y) # obviously prints 4, 4
y = 5
print(x, y) # what about this?
```

1. *Passing By Reference.* The first interpretation is that by setting `y = 5`, we are modifying the value that `y` points to be `5`. Since the pointer `x` also points to the same memory address pointed by `y`, then `x` also should equal `5`.
2. *Passing By Value.* By setting `y = 5`, we create a new `int` object, reassign the pointer `y` to the new object. Therefore `x` still points to `4` and `y` now points to `5`.

```c
// Pass by Reference
int* x_ = malloc(sizeof(int));
*x_ = 4;
int** x = &x_;
int** y = &x_;
printf("%d, %d\n", **x, **y); // 4, 4

**y = 5;
printf("%d, %d\n", **x, **y); // 5, 5
.
.
```

```c
// Pass by Value
int* x_ = malloc(sizeof(int));
*x_ = 4;
int** x = &x_;
int** y = &x_;
printf("%d, %d\n", **x, **y); // 4, 4

int *y_ = malloc(sizeof(int));
*y_ = 5;
y = &y_;
printf("%d, %d\n", **x, **y); // 4, 5
```

Though Python does not technically use references vs values, this analogy is helpful to think about.

Seeing as how an integer is immutable and a list is mutable, let's look at how it affects them.

```python
x = 4
print(x, id(x)) # 4 4374664384
x = x + 1
print(x, id(x)) # 5 4374664416
```

```python
y = [1, 2]
print(y, id(y))  # [1, 2] 4340042048
y.append(3)
print(y, id(y)) # [1, 2, 3] 4340042048
```

As we see, we rebind for immutable types, which changes the pointing memory address, and mutate for mutable types, which doesn't change the address. Therefore, if an object is mutable, then we can mutate it.

Example 5.2 (Warning)

This is very subtle and implementation specific. For immutable types, we are pretty much guaranteed rebinding, but for mutable types, we may not be so sure.

1. If we instantiate two lists and concatenate them using `+` into a list with a new name, we call the `__add__` method, which creates a new list object and binds it to that new list.

```python
y = [1, 2]
z = [3]
print(y, id(y))  # [1, 2] 4380248384
print(z, id(z))  # [3] 4380250176
a = z + y
print(a, id(a))  # [1, 2, 3] 4380551424

a[1] = 4
print(a) # [3, 4, 2]
```

```
print(y) # [1, 2]
print(z) # [3]
```

2. If we instantiate two lists and extend them using `+=`, then we call the `__extend__` method, which extends `z` with a copy of `y`. Note that `z[1:]` and `y` are two different lists objects in memory, not the same reference.

```
y = [1, 2]
z = [3]
print(y, id(y))  # [1, 2] 4380248384
print(z, id(z))  # [3] 4380250176
z += y
print(z, id(z))  # [3, 1, 2] 4380250176

z[2] = 9
print(y) # [1, 2]
print(z) # [3, 1, 9]
```

3. Just to see an example of an immutable type, even using the `iadd` method does not keep its original memory address. The entire thing is always allocated to new memory.

```
x = "Hello "
print(id(x)) # 4382416384
print(x)     # Hello
x += "World"
print(id(x)) # 4382723056
print(x)     # Hello World
```

This explains a lot of the weird phenomena, and it is extremely important to know whether a variable is copied by reference or by value, since you'll be able to predict the behavior on one variable if you modify the other one. The common immutable types in Python are string, int, float.

Example 5.3 ()

To drive the point home, take a look at this. T

```
# Pass by value
x = 4
y = x
# Points to same address
print(id(x)) # 4382741696
print(id(y)) # 4382741696
x += 1
# Now it doesn't
print(x)    # 5
print(y)    # 4
```

```
# Pass by reference
x = []
y = x
# Points to same address
print(id(x)) # 4383459648
print(id(y)) # 4383459648
x.append(1)
# Still points to same address
print(x)     # [1]
print(y)     # [1]
```

Example 5.4 (Common Traps)

To initialize a list of zeros, we can just do

```
>>> x = [0] * 5
>>> x[0] = 1
```

```
>>> x
[1, 0, 0, 0, 0]
```

This is all good since primitive types are immutable, so modifying one really just rebinds it to another value and doesn't affect the others. However, if we are initializing a list of lists, then we get something different.

```
>>> x = [[]] * 5
>>> print(x)
[[], [], [], [], []]
>>> x[0].append(1)
>>> x
[[1], [1], [1], [1], [1]]
```

This is because we are instantiating 5 names that all point to the same empty list. Modifying one really is an act of mutating, leading to the changes persisting across all names. This is because the inner list is multiplied and therefore copied *by reference*. This means that all the lists are simply pointing to the same object in memory, and modifying one modifies all.

5.2 Assignments are Everywhere

Let's look at a few more examples where assignment are, starting with enhanced for loops.

Theorem 5.1 (Assignments in Enhanced For Loops)

Enhanced for loops of form `for elem in x` is really an assignment of `elem` to each element of `x`. All of the following are assignments.

```
for elem in ...
[... for elem in ...]
(... for elem in ...)
{... for elem in ...}
```

Take a look at this anomaly.

```
x = [1, 2, 3]
for elem in x:
    elem += 1
print(x) # [1, 2, 3]
```

With the above theorem, the problem is clear. In the first iteration, we have `elem = 1` and `x[0] = 1`. `elem` has been incremented with `iadd` and therefore is rebound to 2, but this does not affect `x[0]`, leading to no changes. Note that if the elements were mutable, then we can make these changes persist.

```
x = [[1], [2], [3]]
for elem in x:
    elem[0] += 1
print(x) # [[2], [3], [4]]
```

In here, `elem` and `x[0]` are bound to `[1]` and have the same memory address. I then access the memory address of the first element of `elem` and rebind it to its increment. While the `1` changes to a `2`, and `elem[0]` points to a different memory address, the memory address of `elem[0]` itself does not change! Therefore, we have effectively changed the value of the element and have basically mutated the array using the `setitem` dunder method.

This also persists in functions as well.

Theorem 5.2 (Assignments in Functions)

Arguments in functions are also assigned, in local scope of course.

Compare these two snippets.

```python
def augment_twice(a_list, val):
  a_list.append(val)
  a_list.append(val)

nums = [1, 2, 3]
augment_twice(nums, 4)
print(nums)          # [1, 2, 3, 4, 4]
```

```python
def augment_twice_bad(a_list, val):
  a_list = a_list + [val, val]

nums = [1, 2, 3]
augment_twice_bad(nums, 4)
print(nums)          # [1, 2, 3]
.
```

1. In the LHS, **nums** is bound to `[1, 2, 3]`. In the function scope, `a_list` is also bound to the same list. We augment 4 twice, which mutates the object, and upon returning, the name `a_list` is removed. However, the changes persist and is seen by `nums`.
2. In the RHS, `nums` is also bound to `[1, 2, 3]`. In the function, `a_list` is being rebound since we use the `add` method, effectively creating a new list in memory. Now the two variables point to different objects with different memory addresses, and when the function returns, the new list is deleted. Note that this could be avoided if we use the `iadd` dunder method, which leads to the memory address being preserved.

5.3 Object Caching

In general, if we initialize two variables to be the same value, they do not point to the same memory address.

```python
# Example of when two variables are
# initialized to be the same value, but
# do not point to the same memory
x = 1000
y = 1000
print(id(x)) # 4385025360
print(id(y)) # 4385026288
.
.
.
```

```c
int* x_ = malloc(sizeof(int));
*x_ = 1000;
int** x = &x_;

int* y_ = malloc(sizeof(int));
*y_ = 1000;
int** y = &y_;

printf("%p\n", *x); 0x600001be8040
printf("%p\n", *y); 0x600001be8050
```

However, we can initialize `y` to be equal to `x`, which tells it to point to the same memory address as `x` is, thus having the same `id`.

```python
x = 1000
y = x
print(id(x)) # 4303203888
print(id(y)) # 4303203888
.
.
.
.
```

```c
int* x_ = malloc(sizeof(int));
*x_ = 1000;
int** x = &x_;

int** y = &x_;

printf("%p\n", *x); 0x600002368040
printf("%p\n", *y); 0x600002368040
```

This does not change for mutable types either.

```python
x = []
print(id(x)) # 4378741056
x = []
print(id(x)) # 4378742848
```

Usually, just setting the values equal does not have it point to the same memory address, but for integers `[-5, 256]`, Python caches these numbers so that even if we initialize two numbers with the same integer value, they will always point to the same address.

```python
# Don't need to set y = x
x = 200
y = 200
print(id(x)) # 4314934592
print(id(y)) # 4314934592
```

This is a CPython-specific fact that you should be aware of.

5.4 Default Arguments are Evaluated when Function is Defined

We are used to writing functions with default arguments. An important implementation detail is that default arguments are evaluated when a function is *defined*, not when it is called. Consider the following buggy example.

```python
def stuff(x = []):
    x.append(3)
    print(x)

stuff() # [3]
stuff() # [3, 3]
```

There are two unexpected errors with this:

1. We would expect the second call to `stuff` to print `[3]`.
2. The list that `x` references to should be garbage collected (more on this later) when the name has been deleted after the function returned, but it did not.

We will address this first problem. It turns out that the default argument `[]` is created in memory and every call with the default argument assigns `x` to this same list object in the same address. That is, no new lists are created.

This is of course not a problem if default arguments are immutable types likes integers. Even though the default argument is bound to the same object in memory for all calls, the value cannot be modified since you can only rebind it to another object, so it will not contaminate other calls.

5.5 Item Assignment with Walrus Operator

Avoids Repeated Computation

6 Loops

Iterables, Iterators, Generators, zipping, range vs xrange. Range is an iterable, not iterator.

For loops and while loops are straightforward enough, but it's important to know the difference between them.

6.1 While Loops

In while loops, the condition is rechecked and thus any functions called during this is recomputed at each loop, and so when deleting things from a list, the loop already accounts for the new length. However, a for loop evaluates the length of the list only once and leads to index violation errors.

```
x = [1, 2, 3, 4]
print(x)
i = 0
while i < len(x):
    print(len(x))
    if x[i] == 2:
        del x[i]
    i += 1
print(x)

[1, 2, 3, 4]
4
4
3
[1, 3, 4]
```

```
x = [1, 2, 3, 4]
print(x)

for i in range(len(x)):
    print(i, x[i])
    if x[i] == 2:
        del x[i]
print(x)

[1, 2, 3, 4]
0 1
1 2
2 4
IndexError: list index out of range
.
```

This can also be a problem when evaluating to a list where you may need to append more elements to it. Here we use the previous initial list. We want to append 5 and 6 since 2 and 4 are even, but the extra 6 added will require us to add 7 as well. In a for loop, this also breaks down. The for loop only accounts up to the length of the original list, which will end with 6 as the last element added. Whether you want the condition to by dynamically evaluated at every loop depends on the problem.

```
x = [1, 2, 3, 4]
print(x)

i = 0
while i < len(x):
    print(x[i])
    if x[i] % 2 == 0:
        x.append(max(x) + 1)
    i += 1

print(x)

[1, 2, 3, 4]
[1, 2, 3, 4, 5, 6, 7]
```

```
x = [1, 2, 3, 4]
print(x)

for i in range(len(x)):
    if x[i] % 2 == 0:
        x.append(max(x) + 1)

print(x)

[1, 2, 3, 4]
[1, 2, 3, 4, 5, 6]
.
.
.
```

6.2 Iterators and Iterables

Great, so while loops are conceptually simple in that they simply recompute the condition at each loop. For loops—on the other hand—behave quite differently.

Definition 6.1 (Iterables and Iterators)

An **iterator** class is any class that implements a `__next__()` instance method that either returns some value or raises a `StopIteration`. An **iterable** class is any class that implements a `__iter__()` instance method returning an iterator object. When we use a for loop by saying `for elem in object: ...`,

1. the `object` must be an iterable.
2. the for loop implicitly calls `object.__iter__()` before the loop starts to return an iterator `iter`.
3. the loop will continue to call `iter.__next__()` and assign it to `elem` until a `StopIteration` is raised.

The built-in `iter()` method calls `__iter__()` and `next()` calls `__next__()`. Therefore, the two implementations of the for loop is exactly the same.

```
x = [1, 2, 3, 4]
for elem in x:
    print(elem)
.
.
.
.
.
```

```
x = [1, 2, 3, 4]
x_ = iter(x)
while True:
  try:
    item = next(x_)
  except StopIteration:
    break
  print(item)
```

Therefore, we are really just creating an iterator object around the list and doing a while loop. So a for loop is really just a while loop in the backend!

Everything that you can call a for loop on is an iterator.

```
In [1]: iter("hello")
Out[1]: <str_ascii_iterator at 0x1051d4910>

In [2]: iter([1, 2, 3])
Out[2]: <list_iterator at 0x1051fbb80>

In [3]: iter(range(4))
Out[3]: <range_iterator at 0x10528d6e0>

In [4]: iter({"a" : 1, "b" : 2})
Out[4]: <dict_keyiterator at 0x10519f6f0>
```

A common mistake to confuse iterables with iterators! Note that lists and ranges are *not* iterators! They are iterables, so you must call `iter()` on them before calling `next()`.

```
TypeError                             Traceback (most recent call last)
Cell In[5], line 1
----> 1 next([1, 2, 3])

TypeError: 'list' object is not an iterator

In [6]: next(range(4))
---------------------------------------------------------------------
TypeError                             Traceback (most recent call last)
Cell In[6], line 1
----> 1 next(range(4))

TypeError: 'range' object is not an iterator
```

Now let's implement our own class. There are two ways that we can do this: implement the iterator and iterable in two separate classes, or have 1 class support both `__iter__()` and `__next__()` methods to make it *both* an iterator and iterable.

Theorem 6.1 (Separate Implementations of Iterator and Iterable)

Observe that the state of the `StudentIter` created by each of the two for loops are independent with their own states. Therefore, each of the two `x` that we iterate over are two distinct `StudentIter` object, and so we can hit all 4×4 combinations.

```python
class Student:

  def __init__(self):
    ...

  def __iter__(self) -> "StudentIter":
    """A reusable iterator object"""
    return StudentIter(self)

class StudentIter:

  def __init__(self, student: Student):
    self.student = student
    self.i = -1

  def __next__(self):
    self.i += 1
    if self.i > 3:
      raise StopIteration
    return self.i
```

```
In [14]: for i in x:
    ...:   for j in x:
    ...:     print(i, j)
    ...:
0 0
0 1
0 2
0 3
1 0
1 1
1 2
1 3
2 0
2 1
2 2
2 3
3 0
3 1
3 2
3 3
```

Theorem 6.2 (One Class as Iterator and Iterable)

In this case, the state of the next value returned by `__next__()` is stored in the `Student` object, and so `x` is the one `Student` object.

```python
class Student:

  def __init__(self):
    self.i = -1

  def __iter__(self):
    "Nonreusable iterator object"
    return self

  def __next__(self):
    self.i += 1
    if self.i > 3:
      raise StopIteration
    return self.i
```

```
In [13]: x = Student()
In [14]: for i in x:
    ...:   for j in x:
    ...:     print(i, j)
    ...:
0 1
0 2
0 3
.
.
.
.
.
.
```

Example 6.1 (Common Trap)

Look at the following code

```
>>> x = [1, 2, 3, 4]
>>> for elem in x:
...     elem += 1
...
>>> x
[1, 2, 3, 4]
```

This is clearly not our intended behavior. This is because in the backend, the `elem` is really being returned by calling `next()` on the iterator object. The type being returned is an `int`, a primitive type, and therefore it is passed *by value*. Even though `elem` and `x[i]` points to the same memory address, once we reassign `elem += 1`, elem just gets reassigned to another number, which does not affect `x[i]`. Note that this does not work as well since `elem` is just being copied by value and not by reference, and again further changes to `elem` will decouple it from `x[i]`.

```
>>> x = [1, 2, 3, 4]
>>> for i, elem in enumerate(x):
...     elem = x[i]
...     elem += 1
...
>>> x
[1, 2, 3, 4]
```

To actually fix this behavior, we must make sure to call the `__setitem__(i, val)` method, which can be done as such.

```
>>> x = [1, 2, 3, 4]
>>> for i in range(len(x)):
...     x[i] += 1
...
>>> x
[2, 3, 4, 5]
```

Note that if we had nonprimitive types in the list, then the iterator will copy by reference, and we don't have this problem.

```
>>> x = [[1], [2], [3]]
>>> for elem in x:
...     elem.append(4)
...
>>> x
[[1, 4], [2, 4], [3, 4]]
```

Another fact about `range` is that it is *lazy*, meaning that to save memory, calling `range(100)` does not generate a list of 100 elements. The iterator really evaluates the next number on demand, which adds runtime overhead but saves memory.

6.3 Generators

With iterators, we can cleverly keep track of states to design a custom behavior of looping, and as we have seen with range objects, we can also reduce memory by using lazy evaluation. One disadvantage is that there is relatively a lot of boilerplate code to design such an iterator. This is where generators come in.

Definition 6.2 (Generator)

A **generator function** is a function that returns a both an iterable and iterator object (so has its own `__iter__()` and `__next__()` method with the `yield` keyword). The following are equivalent.

```python
# generator function
def make_counter(max):
  count = 1
  while count <= max:
    yield count
    count += 1

counter = make_counter(5)
.
.
```

```python
class Counter:
  def __init__(self, max):
    self.max = max
    self.count = 0

  def __iter__(self):
    return self

  def __next__(self):
    if self.count < self.max:
      self.count += 1
      return self.count
    else:
      raise StopIteration

counter = Counter(5)
```

By default, you should always try to use generators over iterators, and change to the latter if either

1. the state you are maintaining over the loop is complex, or
2. the loop needs to be reusable.

7 Function Closures and Variable Scopes

Therefore, this can lead to buggy behavior when using mutable types where it may be passed by reference.

Nonlocal and global keywords.

8 Composing Classes

If you find yourself nesting built-in types, this is prob an indicator to compose classes. @dataclass.dataclass operator to define simple data structures.

9 Decorators

Note that in Python, functions are first-class citizens, which means three things:

1. They can be treated as objects.

```python
def shout(text):
  return text.upper()

print(shout('Hello'))  # HELLO
yell = shout
print(yell('Hello'))   # HELLO
```

2. They can be passed into another function as an argument.

```python
def shout(text):
  return text.upper()

def whisper(text):
  return text.lower()

def greet(func):
  greeting = func("Hi, How are You.")
  print (greeting)

greet(shout)    # HI, HOW ARE YOU.
greet(whisper)  # hi, how are you.
```

3. They can be returned by another function.

```python
def create_adder(x):
  def adder(y):
    return x+y

  return adder

add_15 = create_adder(15)
print(add_15(10)) # 25
```

Say that you have a function `f` that does something. I want to modify the behavior so that I do something either before of after `f` is called automatically, but I don't want to manually add code into the function body. What I can do is simply define another function `wrapper` and call `f` inside it.

```python
def f():
    print("Hello world")

def wrapper():
    print("started")
    f()
    print("ended")

wrapper() # "started\n Hello world\n ended"
```

Great, we can do this for one function. But what if there were thousands of functions I want to do this for? Rather than creating a wrapper function for each function, I can make a third function called `decorator` that takes in the original function `f` and outputs the `wrapper` function.

```python
def decorator(f):
  def wrapper():
    print("started")
    f()
    print("ended")

  return wrapper

def f():
  print("Hello world")

wrapper = decorator(f)
wrapper() # "started\n Hello world\n ended"

decorator(f) # <function decorator.<locals>.wrapper at 0x100b38e00>
decorator(f)() # "started\n Hello world\n ended"
```

This way, I can modify any function I want with this behavior, and is known as *function aliasing*. This is essentially what a decorator is.

Definition 9.1 (Decorators)

Decorators are used to modify the behavior of your functions without changing its actual code, used with the operator. The two are equivalent.

```python
def decorator(f):
  def wrapper():
    print("started")
    f()
    print("ended")

  return wrapper

def f():
  print("Hello world")

f = decorator(f)
f() # "started\n Hello world\n ended"
```

```python
def decorator(f):
  def wrapper():
    print("started")
    f()
    print("ended")

  return wrapper

@decorator
def f():
  print("Hello world")

f() # "started\n Hello world\n ended"
```

This means that every time I call the function `f`, it really calls the function `decorator` with `f` passed into it as an argument. With functions that have arguments, the wrapper function should also have the same arguments. Generically, we can just use the `args` and `kwargs` arguments to unpack these variables so that `wrapper`'s arguments always matches those of `f`'s arguments, but we can modify these arguments for extra functionality as well.

```python
# generic args and kwargs
def decorator(f):
  def wrapper(*args, **kwargs):
    print("started")
    f(*args, **kwargs)
    print("ended")

  return wrapper

@decorator
def f(string):
  print(string)

f("Hello World")
# started
# Hello World
# ended
```

```python
# custom arguments
def decorator(f):
  def wrapper(string, start_msg):
    print(start_msg)
    f(string)
    print("ended")

  return wrapper

@decorator
def f(string):
  print(string)

f("Hello World", "time to go")
# time to go
# Hello World
# ended
```

If we want to get the return values of this function, we can store the return value in temporary variable `tmp`, run whatever code after the function `f`, and finally return `tmp` in `wrapper`.

```python
def decorator(f):
    def wrapper(*args, **kwargs):
        print("started")
        tmp = f(*args, **kwargs)
        print("ended")
        return tmp

    return wrapper

@decorator
def f(string):
    return string + "!"

print(f("Hello World"))
# started
# ended
# Hello World!
```

Example 9.1 (Measuring Total and CPU Runtime)

If we want to find the runtime of a function, we can do this easily.

```python
import time

def runtime(f):
  def wrapper(*args, **kwargs):
    start = time.time()
    product = f(*args, **kwargs)
    end = time.time()
    print(f"Took {end - start} s")
    return product
  return wrapper

```

```python
@runtime
def dot(list1, list2):
  res = 0
  for x, y in zip(list1, list2):
    res += x * y
  return res

x = [1, 2, 3]
y = [2, 2, 3]
result = dot(x, y)  # Took 3.814697265625e-06 s
print(result)       # 15
```

However, this is not accurate as the OS will switch between different processes. Therefore, the process time is more accurate.

```python
import numpy as np
import time

def cpu_usage(f):
  def wrapper(*args, **kwargs):
    start_cpu = time.process_time()
    result = f(*args, **kwargs)
    end_cpu = time.process_time()
    print(f"CPU time: {end_cpu - start_cpu:.6f} seconds")
    return result
  return wrapper

@cpu_usage
def matrix_mult(a, b):
  return np.matmul(a, b)

x = np.random.randn(2000, 2000)

matrix_mult(x, x) # CPU time: 0.772730 seconds
```

Example 9.2 (Memory Usage)

We can measure memory usage with the `psutil` library.

```python
import numpy as np
import psutil, os

def memory_usage(f):
  def wrapper(*args, **kwargs):
    process = psutil.Process(os.getpid())
    mem_before = process.memory_info().rss
    result = f(*args, **kwargs)
    mem_after = process.memory_info().rss
    print(f"Memory usage: {(mem_after - mem_before) / 1024 / 1024:.2f} MB")
    return result
  return wrapper

@memory_usage
def matrix_mult(a, b):
  return np.matmul(a, b)
```

```python

x = np.random.randn(2000, 2000)
matrix_mult(x, x) # Memory usage: 46.81 MB
```

Example 9.3 (Measuring Function Call Count)

To measure how many times a function has been called, we can use the decorator.

```python
def call_counter(f):
    def wrapper(*args, **kwargs):
        wrapper.count += 1
        print(f"Function '{f.__name__}' called {wrapper.count} times")
        return f(*args, **kwargs)
    wrapper.count = 0
    return wrapper

@call_counter
def factorial(x):
    if x == 1:
        return 1
    return x * factorial(x - 1)

result = factorial(7)
# Function 'factorial' called 1 times
# Function 'factorial' called 2 times
# Function 'factorial' called 3 times
# Function 'factorial' called 4 times
# Function 'factorial' called 5 times
# Function 'factorial' called 6 times
# Function 'factorial' called 7 times
print(result)
# 5040
```

functools.wraps.

10 Raising Exceptions

Many beginners prefer to return None, but you should really be raising exceptions.

11 Package Management

12 Inspect

`inspect` is a module that allows you to get live information about live objects such as modules, classes, and functions.

Definition 12.1 (`getsource`)

The `getsource` method allows you to see the text of live objects.

```
>>> import inspect
>>> backbone_module = construct_backbone('resnet50[pretraining=inaturalist]')
>>> model = backbone_module.embedded_model
>>> print(inspect.getsource(model.forward))
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        return x

>>> print(inspect.getsource(model.__class__))
class ResNet_features(nn.Module):
    """
    the convolutional layers of ResNet
    the average pooling and final fully convolutional layer is removed
    """

    def __init__(self, block, layers, num_classes=1000, zero_init_residual=False):
        super(ResNet_features, self).__init__()
        ...
    ...
```

Figure 7: Say that you have some torch model that is either inaccessible or is hidden away through so many imports that you have a hard time accessing it. Rather than going through several files and having to parse which methods are relevant, is overwritten, or called, you can just inspect the methods and classes directly.