Data Structures in Python

their properties and their use cases

Sukrit Gupta and Nitin Auluck

February 9, 2024

Acknowledgement and disclaimer

All mistakes (if any) are mine.
I have used several other sources which I have referred to in the appropriate places.

Outline

1. Intuition behind structured data types

2. Some key concepts in Python
   - Behind the scenes
   - Python Objects
   - Memory management in Python
   - Function Calls

3. Planting the seeds for computational complexity

Section 1

Intuition behind structured data types

Structured data types in Python

- Lists/Arrays, Dicts, Sets and Tuples are the structured data types in Python.
- Why don't we have a single structured data type that is universal?
- I want you to understand why different structured data types exist.
- Let us link this to humans.

Example 1: Different humans, different talents

Figure: Jerry Seinfeld [1]

Great comedian/actor, but would you go to him for investment advice?

[1] tvline.com

Example 2: Different humans, different talents

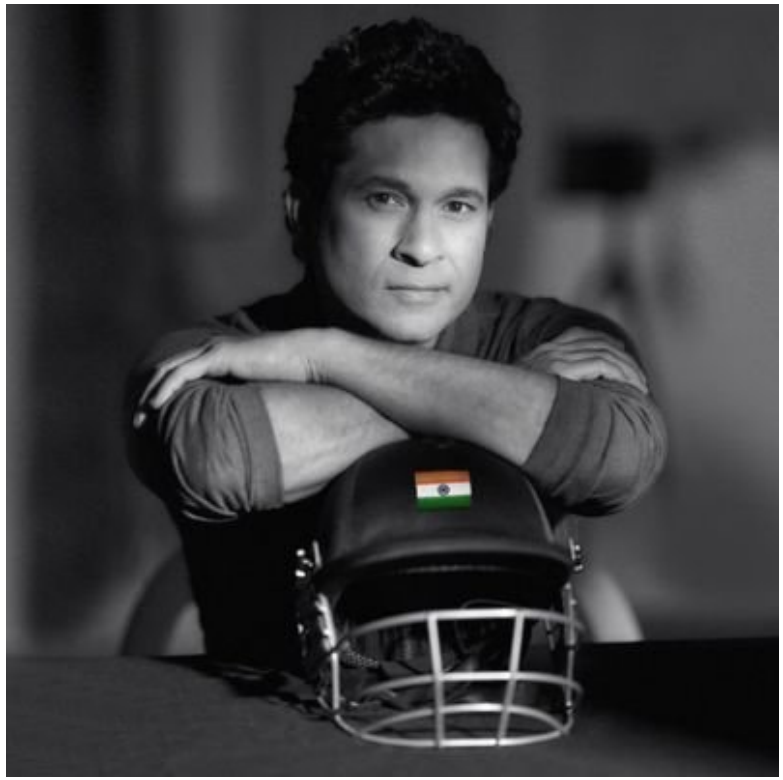


Figure: Sachin Tendulkar[2]

Great cricketer, but would he be a very successful academician?

[2] twitter.comsachin_rt

Example 3: Different humans, different talents

Figure: Kangana Ranaut [3]

Great actress, but would you trust her to write accurate/unbiased history books?

[3] theweek.in

What am I trying to tell you?

- There is a use-case/domain for each person (structured data type).
- We should learn to put them in places where they fit and will perform well.
- In other places, they will lead to *suboptimal* performance.
- Before we move to structured data types, it is important to get hold of a few key concepts in Python.

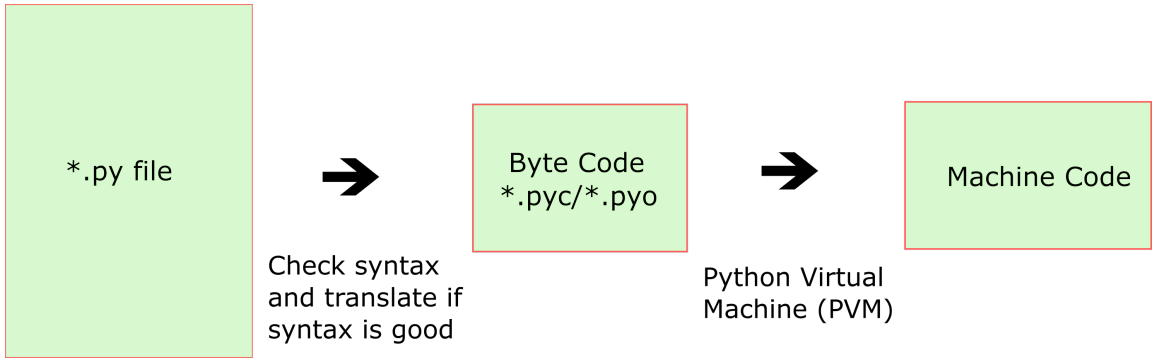
Section 2

Some key concepts in Python

2.1 Behind the scenes

What happens when you run a program in Python?

With C/C++, when you compile the code, it is transformed to a hardware readable form known as *machine code*. However, this is not how it happens with Python.



- Step 1: The python compiler reads a python source code or instruction. Then it checks the syntax of each line. If it encounters an error, it immediately halts the translation and shows an error message.
- Step 2: If there is no error, i.e. if the python instruction or source code is well-formatted then the compiler translates it into its equivalent form in an intermediate language called "Byte code" (*.pyc* or *.pyo*).
- Step 3: Byte code is then sent to the Python Virtual Machine (PVM) which is the python interpreter. PVM converts the python byte code into machine-executable code. If an error occurs during this interpretation then the conversion is halted with an error message.
- For details on how byte code looks like and how it is implemented, refer to: Article

2.2 Python Objects

Objects in Python

- Python is an object oriented programming (OOP) language.
- Everything in Python is an object (including variable, function, list, tuple, dictionary, set, et cetera). For example: an integer variable belongs to the integer class. We will go in depth of the concept of the class and the object as we move on.
- Every object has an *identity*, a *type*, and a *value*.

- Identity: An object's identity never changes once it has been created; you may think of it as the object's address in memory. The is operator compares the identity of two objects; the id() function returns an integer representing its identity.
- Type: An object's type defines the possible values and operations (e.g. "does it have a length?") that type supports. The type() function returns the type of an object. An object type is unchangeable like the identity.
- Value: This is the value that is attached to the object.

Containers

Some objects contain references to other objects, these objects are called containers. Some examples of containers are a tuple, list, set, and dictionary.

2.3 Memory management in Python

References in Python

A reference is a name that we use to access a data value (i.e., an object).
The most famous references in programming are variables. Let's see what happens when you do something like x = 300.

- What is the reference here? x
- What is the data value here? 300
- First, the value (more accurate an integer object) 300 is assigned to a memory address.
- Then, the reference x *points to the memory address* (and not the value).

Reference does not point to a value in Python but points to the memory address of an object.

Therefore, x = 300, the reference x is pointing to a memory address that the integer object 300 is stored.

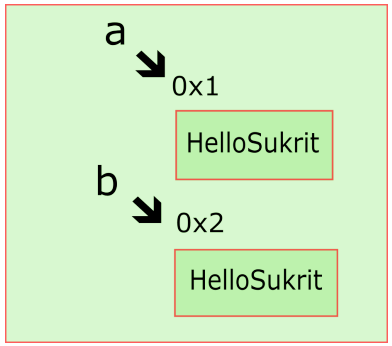
```
>>> x = 300
>>> id(x)
140613407592426
>>> hex(id(x)) #In CS, we show mem addresses in hex numbers
'0x7FE31C35CFEA' #prefix 0x is used in CS to indicate a number in hex
```

When we run the first line x = 300, Python stores integer object 300 in a memory address 140613407592426 on my computer (different from yours). After storing the int object 300 in memory, Python tells the reference (or variable) x to memorize this address (140613407592426 or 0x7FE31C35CFEA) as its value.

Usual references in Python

```
>>> a = "HelloSukrit"
>>> b = "HelloSukrit"
>>> hex(id(a)) #
>>> hex(id(b)) #
```

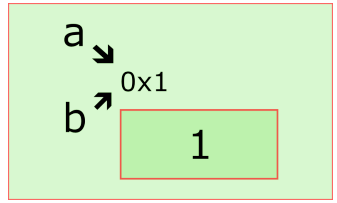
Returns different addresses for the variables.



Let's look at another example.

Interning in Python

```
>>> x = 1
>>> y = 1
>>> hex(id(x))
'0x7ffdf176a190'
>>> hex(id(y))
'0x7ffdf176a190'
```



Surprisingly, the memory addresses that both variables point to are the same.

Python does a process called “interning”. For some objects, Python only stores one object in the memory and ask different variables to point to this memory address if they use those objects.

Python does interning on integer numbers, boolean, and some strings. The rules for interning objects change with Python versions.

Mutable and Immutable objects

- Mutable is when something is changeable or has the ability to change.
- In Python, 'mutability' is the ability of objects to change their values. The mutability of an object is determined by its type.
- Some of the mutable data types in Python are list, dictionary, set and user-defined classes.
- On the other hand, some of the immutable data types are int, float, decimal, bool, string, tuple, and range.

Examples: Mutability in List vs Tuple

```
>>> x = ['apple', 'banana', 'cherry']
>>> id(x)
140030422603200
>>> x.append(1)
>>> id(x)
140030422603200
>>> x
['apple', 'banana', 'cherry', 1] #
>>> x = ("apple", "banana", "cherry")
>>> id(x)
140030420896256
>>> y = (1,)
>>> x += y
>>> x
('apple', 'banana', 'cherry', 1)
>>> id(x)
140030420855360
```

2.4 Function Calls

Functions

```
def func_1(a, b):
    ...
    return ...

x = ...
y = ...
func_1(x, y)
```

x, y are called *actual* parameters and a, b are called *formal* parameters.

What happens when you call a function with parameters in Python? Let's see.

```
def func_1(a, b):
    ...

func_1(x, y)
```

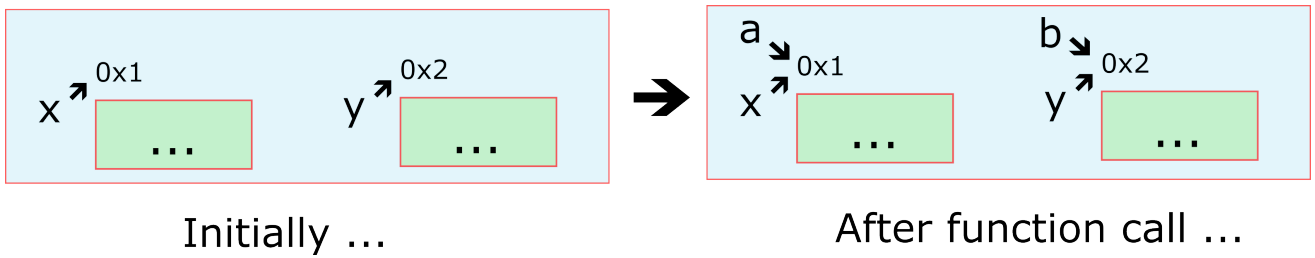


Figure: Memory allocation during function call

Does someone see the problem?

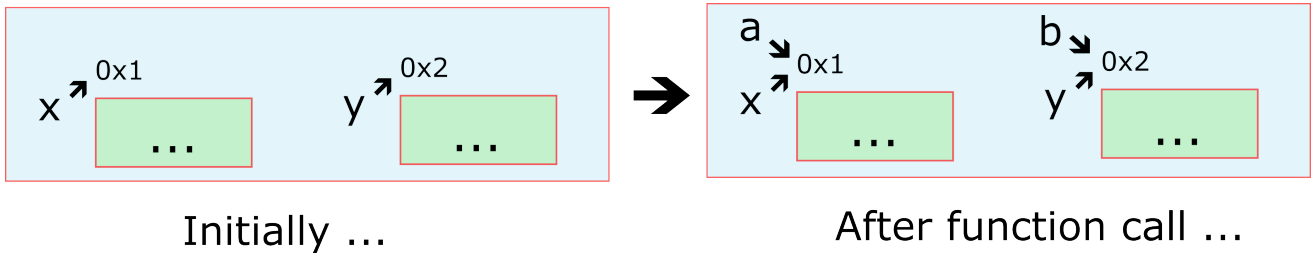


Figure: Memory allocation during function call

Suppose x referenced an immutable object and y referenced a mutable object.

And we update both a and b within the function. What happens next?

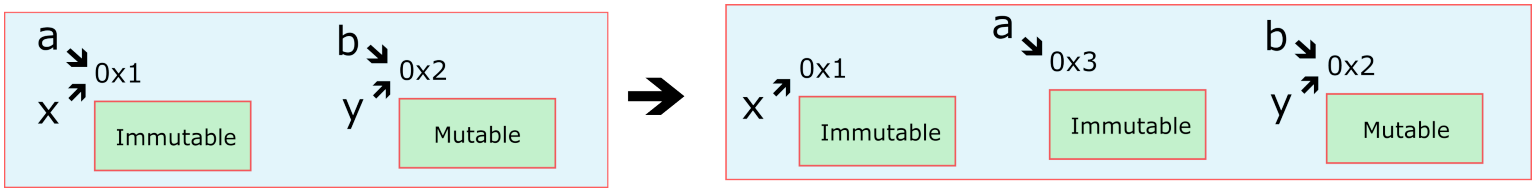


Figure: Mutable objects are altered in place.

What does this mean when you code?

Mutable objects in function call

```
def func(a, b):
    a +=1
    b.append(4)
    return

x = 5 #Immutable
y = [1, 2, 3] #Mutable
func(x, y)
print(x) # 5
print(y) # [1, 2, 3, 4]
```

The mutable object is altered in memory. Need to exercise caution while using mutable objects in function call statements.

In order to avoid the above, pass a copy of the original mutable object while calling the function.

Section 3

Planting the seeds for computational complexity

Intuition using an example

Let us say that you are asked to write a program that finds an approximate solution to a problem, i.e. the computed solution is "close enough" to the actual solution.
Let's say that you have to find an approximation to the square root of a number, how do you proceed?

Solution 1: Exhaustive search

Brute force search.

```
x = 25
epsilon = 0.01
step = 0.0001
num_guesses = 0
approx_ans = 0.0

while abs(approx_ans**2 - x) >= epsilon and approx_ans*approx_ans <= x:
        approx_ans = approx_ans + step
        num_guesses = num_guesses + 1

print ('num_guesses =', num_guesses)

if abs(approx_ans**2 - x) >= epsilon: #not close to actual solution
        print ('Failed on square root of', x)
else:
        print (approx_ans, 'is close to square root of', x)
```

Exhaustive Search

- What are the number of steps needed for guessing the square root of 25?
- What about 2,50,000?
- This solution quickly becomes intractable for larger numbers.

Thinking about a solution ...

- Please get any textbook in your bag and open it. Let's assume that you have n pages in the book.
- To search a particular page number in the textbook using the exhaustive search technique, I will turn the pages one by one from the start.
- Search for page 1 in the textbook. How many pages did you have to turn?
- Best case: 0.
- Now search for n. How many pages did you have to turn?
- Worst case: n.
- Now search for page $0.75n$. How many pages did you have to turn?
- Now search for page $0.25n$. How many pages did you have to turn?
- What is the average of the four? $\approx n/2$

Now try this ...

To search a particular page number k in the textbook: Open the middle page $n/2$; if page $k > n/2$: open the middle page of the second half $(3/4n)$, else: open the middle page of the first half $(1/4n)$ and so on until you find the page you're looking for.

Thinking about a solution ...

- Search for page 100 in the textbook. How many pages did you have to turn?
- Now search for page 250. How many pages did you have to turn?
- Is this faster?
- Coming back to our problem of approximating the square root of a number ...

Solution 2: Bisection search

```
x = 25
epsilon = 0.01
num_guesses = 0
low = 0.0
high = x
approx_ans = (high + low)/2.0
while abs(approx_ans**2 - x) >= epsilon:
        print ('low =', low, 'high =', high, 'approx_ans =', approx_ans)
        num_guesses = num_guesses + 1
        if approx_ans**2 < x:
                low = approx_ans
        else:
                high = approx_ans
        approx_ans = (high + low)/2.0
print ('num_guesses =', num_guesses)
print (approx_ans, 'is close to square root of', x)
```

What did we learn today?

1. Intuition behind structured data types

2. Some key concepts in Python
   - Behind the scenes
   - Python Objects
   - Memory management in Python
   - Function Calls

3. Planting the seeds for computational complexity

Thank you!