June 5, 2021

***ETL*:** raw data, clean, EDA, database, back-end, query, ML, algorithm, hard-coded, non hard-coded, dynamic, front-end, UX/UI design, front-end, framework, JS, ETL, pipeline, data ingestion

Python (robot snake): what are "algorithms" and "data structures"?

Algorithms depend on *data structures*. An *algorithm* is a "finite sequence of well-defined, computer-implementable instructions, typically to solve a class of specific problems or to perform a computation".

Take this, it's an algorithm (do NOT try to understand the code, just memorize what it looks like)

```
array = [1, -10, 2, 3, 4]

def BubbleSort(array):

    for i in range(len(array)):
        for j in range(len(array)-i-1):

            if array [j] > array [j+1]:
                array [j], array [j+1] = array [j+1], array [j]
    print(array)

BubbleSort(array)

#Output: [-10, 1, 2, 3, 10]
```

LIAM'S LAB

It's not just any algorithm – it's a sorting algorithm; in other words, it's an algorithm specified to sort.

This algorithm *depends* on a list (or "array" outside of Python's lingo): `[1, -10, 2, 3, 4]` which means it *depends* on data, like the integers `1, -10, 2, 3, 4` that are *structured* as a list, i.e. a "Data Structure" ! Algorithms can depend on all sorts of *data structures*; for instance, take this algorithm that takes in a freaking *Tree* (also called a "graph") for god's sake.

```
graph = {
    '1': ['3', '7'],
    '7': ['1', '2'],
    '3': [],
    '2': ['8'],
    '8': ['3']
}

visited = set()

def DepthFirstSearch(visited, graph, node):

    if node not in visited:
        print(f'Hey I found {node}')
        visited.add(node)
        for neighbor in graph[node]:
            DepthFirstSearch(visited, graph, neighbor)

print('Hey let me climb down this tree, watch me go \n', DepthFirstSearch(visited, graph, '1'))
```

Our first algorithm "sorted" a list and depends on a *list* for its data structure. Our second algorithm "searched" a graph (or *climbed down a tree*) and depends on a *graph* for its data structure.

Programming & computer science has a *lot* of data types. Each language has its own way of referring to things and way of doing things, way of calling things. Each process of DS may require its own algorithm; for instance, if you're web-scraping a website, you might want to know some "Tree" algorithms since websites have branches and stuff. Of course, in interviews, anyone can ask *anything*, which sort of sucks. Don't study this to ace an interview, study this because you want to. Trying to learn everything is the same as electing to live in a black hole. Total waste of time!

Python (robot snake): what "data structures" can we start with?

Start by writing down what *you* think each of things should be called. If you already know the "formal" name for them, come up with something else!

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


{ 'key': 'value' }


"Hello!"


{1, 2, 3}


(1, 2, 3)
```

LIAM'S LAB

Just read this aloud: `[1, 2, 3, 4, 5]` . What does it sound like you're doing? …
Listing things? You'd be right! `[1, 2, 3, 4, 5]` is a "list", or as they're more famously called: "arrays".

`{ 'key': 'value' }` is a little bit more complicated. I'll give you an example. If a `key` is `'fruit'` then what could a `value` be? `Apple` ? Sure! What if a `key` is `'coffee drink'` then what could a `value` be? `Lattee` ? Sure! What if a `key` is `'你好'` or `'안녕하세요'`) then what could a `value` be? `Helloo` ? Sure! What if I had something like:

```
{
        '你好': 'Hello' ,
        '咖啡': 'Coffee' ,
        '跑步': 'To run' ,
        '失败': 'To fail' ,
        '道可道，非常道，名可名，非常名。无名天地知识，有名万物之母': 'One way is a way is not every way;
one name is a name is not every name. Without name everything begins, with a name all things
exist.'

}
```

What does this look like? …. Starts with a "d". A *dictionary*? Yes! This is a dictionary. And this `{ 'key': 'value' }` action is called a "key-value pair", because there's two of them. A *key* is something we only keep one copy of in our lives, so *keys* are unique, but values can be a *list*! Like `'fruit'` could have the `value` `["Apple", "bAnAnA", "O rag man go (orange mango)"]` .

The next thing is `"Hello!"` . Let me write this a different way:

```
H        e          l        l        o          !
```

Can you rewrite "Hello!"? Did you notice how you were taking each *letter* or *character*, `H e l l o !` and putting them next to each other? If you had to name what you were doing – what would you say you were doing with those *characters*?

Yes! You were **stringing** them together. So it's called a "string". It's important to note that if we have something like `"Hello world!"`, if we were to list out what we are stringing together: `["H","e","l","l","o","w","o","r","l","d","!"]` .

I'm missing one thing in my list. Can you figure out what it is?

I'm missing a god dang **space** !

```
list = ["H","e","l","l","o"," ","w","o","r","l","d"]
print(('').join(list))

#Output: Hello world
```

LIAMS LAB

It's important to note that in Python a *space* makes up, or "constitutes" (if you want a big word) its own character.

Bonus grandmaster question: Can you change "Hello world" to be "I know how to change strings"?

Now let's move onto `{1, 2, 3}` . Let me illustrate to you what this does:

```
l = [1, 1, 1, 1, 1, 1, 1]
print(f'There are {l.count(1)} 1s')
#Output: There are 7 1s.

print(set(l))

#Output: {1}
```

What happened here? … What do you think?

Yes, *duplicates* were removed! In programming, we will start to say something went from being *non-unique* to being *unique*. Uniqueness is just like, are there duplicates or not.

Another way of saying this is there's a unique **set** of stuff now. So, it's called a **set**. That's what I redacted up top.

Finally, we have `(1, 2, 3)` . This is pretty hard to guess, so I'm just going to spoil it for you: it's called a **tuple**.

```
#Tuple examples!

()
(1)
(1, 2)
(1, 2, 3)
(1, 2, 3, 4)
```

Tuples can be any length. "Any" sort of sounds like "n". Tuples can be "n" length. "n" just means a finite integer, so you can't have 1.2 length Tuples, just 1, 2, 3 etc.

⚔. DUEL ⚔: Time for a d-d-d-d-d duel.
What's the difference between a `SET`, a `LIST` and a `TUPLE` ?

Rachel, team `LIST`	Ji, team `SET`	Hao, team `TUPLE`
[1, 1, 10, 9, 5, 4]	{1, 4, 5, 9, 10} {10, 5, 9, 4, 1}	(1, 1, 4, 5, 9, 10)
Not ordered		Ordered
Non-unique	Not ordered	Non-unique
Mutable	Unique	Immutable

LIAMs LAB

Without looking at your notes, what are the 5 data types we just went over?

Leaving space for your notes … This is the end of this section! Just one more thing to go over today.

Python (robot snake): what's "iteration" and "loops"?

If I say I'm stuck in the same loop, I mean I keep doing the same thing over and over again. If I say I'm stuck in a loop of negativity, I mean I interpret everything negatively; or if I say I'm stuck in the same loop of making coffee at 8 AM every day, I mean that I make coffee at 8 AM everyday.

Let's run with this coffee example. What if I asked you to count all the beans in a coffee jar? How would you do it? Write it down below:

Notice what's going on in your brain here: you start at 0, and you pick up each bean and go 0+1, 1+1, 2+1, 3+1. Or, if you're hyper-efficient, you might go in "batches" and pick up 5 beans at a time. The point is, you start at 0, and **increment** (count) that "bean" number, **for** every bean **in** the jar

Say this out loud: `for every_bean in the_jar`. This is how computers might approach this same task too.

```
bean_count = 0
the_jar = ['bean', 'bean', 'bean', 'bean', 'bean', 'bean']

for every_bean in the_jar:
    bean_count = bean_count + 1

print(f'There are {bean_count = } beans!')

#Output: There are bean_count = 6 beans!
```

When you're counting every bean, you're **iterating** over every bean. We say you're writing a **for loop** that **iterates** over every_bean in the_jar and **increments** a **counter** (bean_count) by 1 each time.

There are some other kinds of loops we will cover next week. If you want to get a head start, you can look up `while loops`.

Python (robot snake): where can we run this code?

So I have been giving you paragraphs of code this whole time. What do those look like on the page? Paragraphs, sure, but they also look like **blocks**. So we can say I've been giving you **blocks of code** with nowhere to **run** them, or a way of getting those outputs. Just like paragraphs have sentences, **blocks of code** have **lines of code**.

I really do not believe there is one way of doing anything, so there's about 10000 ways you can run code. Some ways that *I* use are **Anaconda, Jupyter notebook** and **Atom**. Anaconda and Jupyter notebook are applications, and Atom is an "IDE" (Integrated Development Environment). There's a bunch of IDEs out there, such as **VS Code, Sublime Text, Spyder, Vim,** yadda yadda. I *strongly* encourage you to find *what you want to use* as part of our goal of self-empowerment.

You can download **Anaconda** online. Use the "graphical installer". After you download that, open Jupyter notebook `demo.ipynb` . What is "`.ipynb`"? Well, the "." is called a **file extension**, like Microsoft Excel is ".xls", programs are ".exe", those are file extensions. The file extension "`.ipynb`" is used for Jupyter notebooks, and can be opened in Jupyter notebook.

Now, we will do the Jupyter notebook demo lab. You can go to demo.ipynb next!

Some notes from that lab we did on June 5, 2021: "Markdown" is for text; "Code" is for "code".
Some notes on shortcuts in Jupyter:

`a`	`Insert new cell above`
`b`	`Insert new cell below`
`esc-m`	`Change cell-type to markdown`
`esc-y`	`Change cell-type to code`
`Shift-Enter`	`Run a cell`
`dd`	`Delete a cell`

LIAM'S LAB

Thanks for coming today! That's all for this week.

Homework:

Finish demo.ipynb !

Bonus grandmaster question:

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

See you next week!