1.4 Data types

`Kiddo explanation 😇:`

We might use many materials like sand, bricks, concrete to construct a house. These are basic and essential needs to have the construction done and each of them have a specific role or usage.

Likewise, we need various data types like string, boolean, integer, dictionary etc.. for the development of a code. We need to know where to use a specific data type and it's functionality. 😊

We have various built-in data types that come out of the box 😎.

Data type	Mutable?
None	❌
bytes	❌
bool	❌
int	❌
float	❌
complex	❌
str	❌
tuple	❌
list	✅
set	✅
dictionary	✅

The First question we would be interested in is "What is Mutable?🤔". If a object can be altered after its creation, then it is Mutable, else Immutable.

None

None is a singleton object, which represents empty or null.

Example of None usage:

In this example, Let's try getting the environment variables 😉

We would be using the `os` module's `getenv` method to fetch the environment variable's value, if there isn't that environment variable, it would be returning `None`

In [1]:
```
import os

# let's set a env variable first
new_environment_variable_name: str = input("Enter the variable name: \n>>>")
new_environment_variable_value: str = input("Enter the variable's value: \n>>
os.environ[new_environment_variable_name] = new_environment_variable_value

# Now let's try to fetch a envrionment's variable value
env_variable_name: str = input("Enter the variable name to be searched: \n>>>
value = os.getenv(env_variable_name)
```

```
if value is None:
    print(f"There is no environment variable named {env_variable_name}")
else:
    print(
        f"The value assigned for the environment variable named {env_variable
    )
```

```
There is no environment variable named Golang
```

bytes

byte objects are the sequences of bytes, these are machine readable form and can be stored on the disk. Based on the encoding format, the bytes yield results.

bytes can be converted to string by decoding it, vice-versa is known as encoding.

bytes objects can be created by prefixing `b` before the string.

In [2]:
```
bytes_obj: bytes = b"Hello Python Enthusiast!"
print(bytes_obj)
```

```
b'Hello Python Enthusiast!'
```

We see that they are visually the same as string when printed. But actually they are ASCII values, for the convenience of the developer, we see them as human readable strings.

But how to see the actual representation of bytes object? 🤔 It's pretty simple 😉! We can typecast the bytes object to a list and we see each character as it's respective ASCII value.

In [3]:
```
print(list(bytes_obj))
```

```
[72, 101, 108, 108, 111, 32, 80, 121, 116, 104, 111, 110, 32, 69, 110, 116, 1
04, 117, 115, 105, 97, 115, 116, 33]
```

bool

bool objects have only two values: `True` ✅ and `False` ❌, integer equivalent of True is 1 and for False is 0

In [4]:
```
do_we_love_python = True
if do_we_love_python:
    print("🐍 Python too loves and takes care of you ❤")
else:
    print("🐍 Python still loves you ❤")
```

```
🐍 Python too loves and takes care of you ❤
```

PS: Boolean values in simple terms mean **Yes** for `True` and **No** for `False`

int

int objects are any mathematical Integers. pretty easy right 😎

In [5]:
```
# Integer values can be used for any integer arithmetics.
# A few simple operations are addition, subtraction, multiplication, divisio
operand_1 = int(input("Enter an integer value: \n>>>"))
operand_2 = int(input("Enter an integer value: \n>>>"))
print(operand_1 + operand_2)
```

float

float objects are any rational numbers.

In [6]:
```
# Like integer objects float objects are used for decimal arithmetics
# A few simple operations are addition, subtraction, multiplication, divisio
# We are typcasting integer or float value to float values explicitly.
operand_1 = float(input("Enter the integer/float value: \n>>>"))
operand_2 = float(input("Enter the integer/float value: \n>>>"))
print(operand_1 + operand_2)
```

```
11.620000000000001
```

complex

complex objects aren't so complex to understand 😉

complex objects hold a Real number and an imaginary number. While creating the complex object, we would be having a `j` beside the imaginary number.

In [7]:
```
operand_1 = 10 + 5j
operand_2 = 3 + 4j
print(operand_1 * operand_2)
```

```
(10+55j)
```

explanation for the above math: 😉

```
math
(3+4j)*(10+5j)
3(10+5j) + 4j(10+5j)
30 + 15j + 40j + 20(j*j)
30 + 15j + 40j + 20(-1)
30 + 15j + 40j - 20
30 - 20 + 15j + 40j
10 + 55j
```

str

string objects hold an sequence of characters.

In [8]:
```
my_string = "🐍 Python is cool"
print(my_string)
```

```
🐍 Python is cool
```

tuple

tuple object is an immutable datatype which can have any datatype objects inside it and is created by enclosing paranthesis `()` and objects are separated by a comma.

Once the tuple object is created, the tuple can't be modified, although if the objects in the tuple are mutable, they can be changed 😊

The objects in the tuple are ordered, So the objects in the tuple can be accessed by using its index ranging from 0 to (number of elements - 1).

In [9]:
```
# tuples are best suited for having data which doesn't change in it's lifetim

apple_and_its_colour = ("apple", "red")
watermelon_and_its_colour = ("watermelon", "green")

language_initial_release_year = ("Golang", 2012)
language_initial_release_year = ("Angular", 2010)
language_initial_release_year = ("Python", 1990)

# We can't add new data types objects, delete the existing datatype objects,
# of the existing objects.

# We can get the values by index.
print(
    f"{language_initial_release_year[0]} is released in {language_initial_rel
)
```

```
Python is released in 1990
```

list

list objects are similar to tuple, the differences are the list object is mutable, so we can add or remove objects in the list even after its creation. It is created by using `[]`.

In [10]:
```
about_python = [
    "interpreted",
    "object-oriented",
    "dynamically typed",
    "open source",
    "high level language",
    "🐍",
    1990,
]
print(about_python)
# We can add more values to the above list. append method of list object is u
# let's give a try 🙃

about_python.append("Guido Van Rossum")
print(about_python)
```

```
['interpreted', 'object-oriented', 'dynamically typed', 'open source', 'high
level language', '🐍', 1990]
['interpreted', 'object-oriented', 'dynamically typed', 'open source', 'high
level language', '🐍', 1990, 'Guido Van Rossum']
```

set

set objects are unordered, unindexed, non repetitive collection of objects. Mathematical set theory operations can be applied using set datatype objects. 😊 it is created by using `{}`.

PS: `{}` denotes a dictionary, we need to use `set()` for creating an empty set, there won't be this issue when creating set objects containing objects, for example: `{1,"a"}`

set objects are good for having the mathematical set operations.

In [11]:
```
set_obj = {6, 4, 4, 3, 10, "Python", "Python", "Golang"}
# We see that we have created a set with 8 objects.
```

```
print(set_obj)
# But when printed, we see that only 6 are present because set doesn't allow
```

```
{'Golang', 3, 4, 10, 6, 'Python'}
```

dict

dictionary objects are used for creating key-value pairs, Here keys would be unique while values can be repeated.

The object assigned to a key can be fetched by using `<dict_obj>[key]` which raises a KeyError when no given key is found. The other way to fetch is by using `<dict_obj>.get(key)` which returns `None` by default if no key is found.

In [12]:

```
dict_datatype = {
    "language": "Python",
    "Inventor": "Guido Van Rossum",
    "release_year": 1991,
}
print(f"The programming language is: {dict_datatype['language']}")
# We could use get method to prevent KeyError if the given Key is not found.
result = dict_datatype.get("LatestRelease")
# Value of the result would be None as the key LatestRelease is not present i
print(f"The result is: {result}")
```

```
The programming language is: Python
The result is: None
```