SUNCORP

Python School

Lesson 4

10 May 2017

PREVIOUSLY ON PYTHON SCHOOL

WE LOOKED AT

FUNCTIONS!

$$x \rightarrow f(x) \rightarrow y$$

```python
def function(x):
    y = x + 1
    return y
```

...

Input → *do something with it* → **Output**

it was a

REVELATION!

simplifying our code-base by
breaking down a *complex* program into simpler parts

↓

many processes can be encapsulated and called on demand we also learnt how to store our functions in

MODULES

which can be `import`-ed as we need them

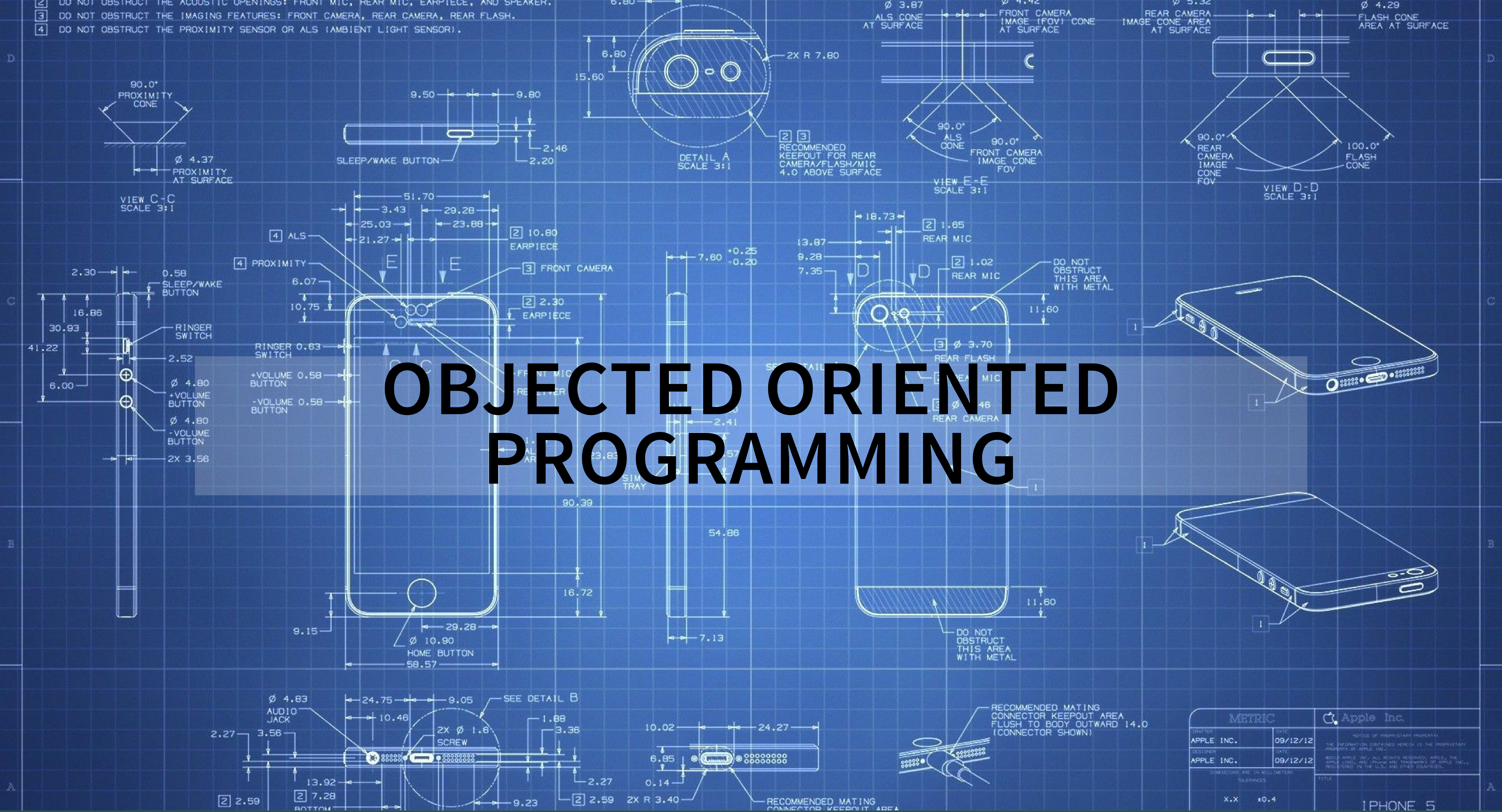
allowing for a more programmatic style

we're now going to put a lot of what we've learnt so far to approach a new concept

one that allows you to package

data-types and **functions**

TOGETHER



OBJECTED ORIENTED PROGRAMMING

Rhys Whitley • Python School • 30 March 2017

WHAT THE HELL IS THAT?

it's programming paradigm that structures data into an abstract collection of attributes and behaviors

TO DEMONSTRATE, LET'S RECALL

STRINGS

Can you recall what makes them unique?

✔ initialised by quotation marks

✔ have size (number of characters)

✔ can be transformed (e.g. uppercase, split, etc.)

✗ but are not mutable

so...

`strings` have a specific behaviour,

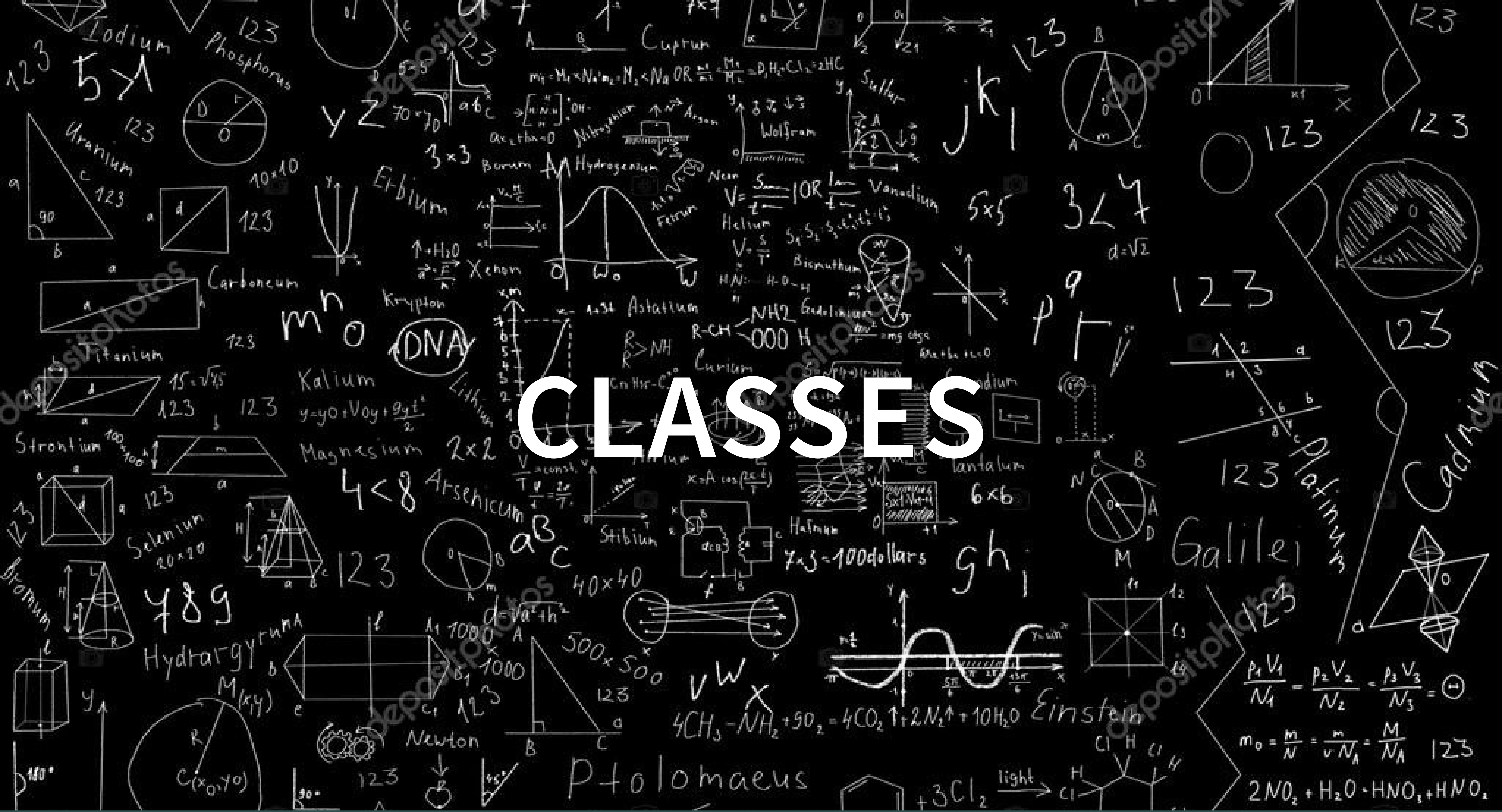
and the data is whatever you've set it to be

```
message = "Go jump in the lake!"
```

Now just hold on to that thought

so let's start with

THE BASICS OF OOP


Iodium
Phosphorus
Cuprum
Sulfur
Uranium
Wolfram
Argon
Nitrogenium
Hydrogenium
Borum
Erbium
Neon
Vanadium
Helium
Ferrum
Bismuthum
Xenon
Carboneum
Krypton
Astatium
Gadolinium
DNA
Titanium
Kalium
Curium
Lithium
Strontium
Magnesium
Tantalum
Arsenicum
Selenium
Stibium
Hafnium
Galilei
Platinum
Cadmium
Bromum
Hydrargyrum
Newton
Ptolomaeus
Einstein
7×3=100dollars
123
4<8
789
$4CH_3-NH_2+9O_2=4CO_2\uparrow+2N_2\uparrow+10H_2O$
$2NO_2+H_2O=HNO_3+HNO_2$
light

CLASSES

CLASSES

can be effectively thought of as a blueprint that describes

STATE	BEHAVIOUR
data	methods

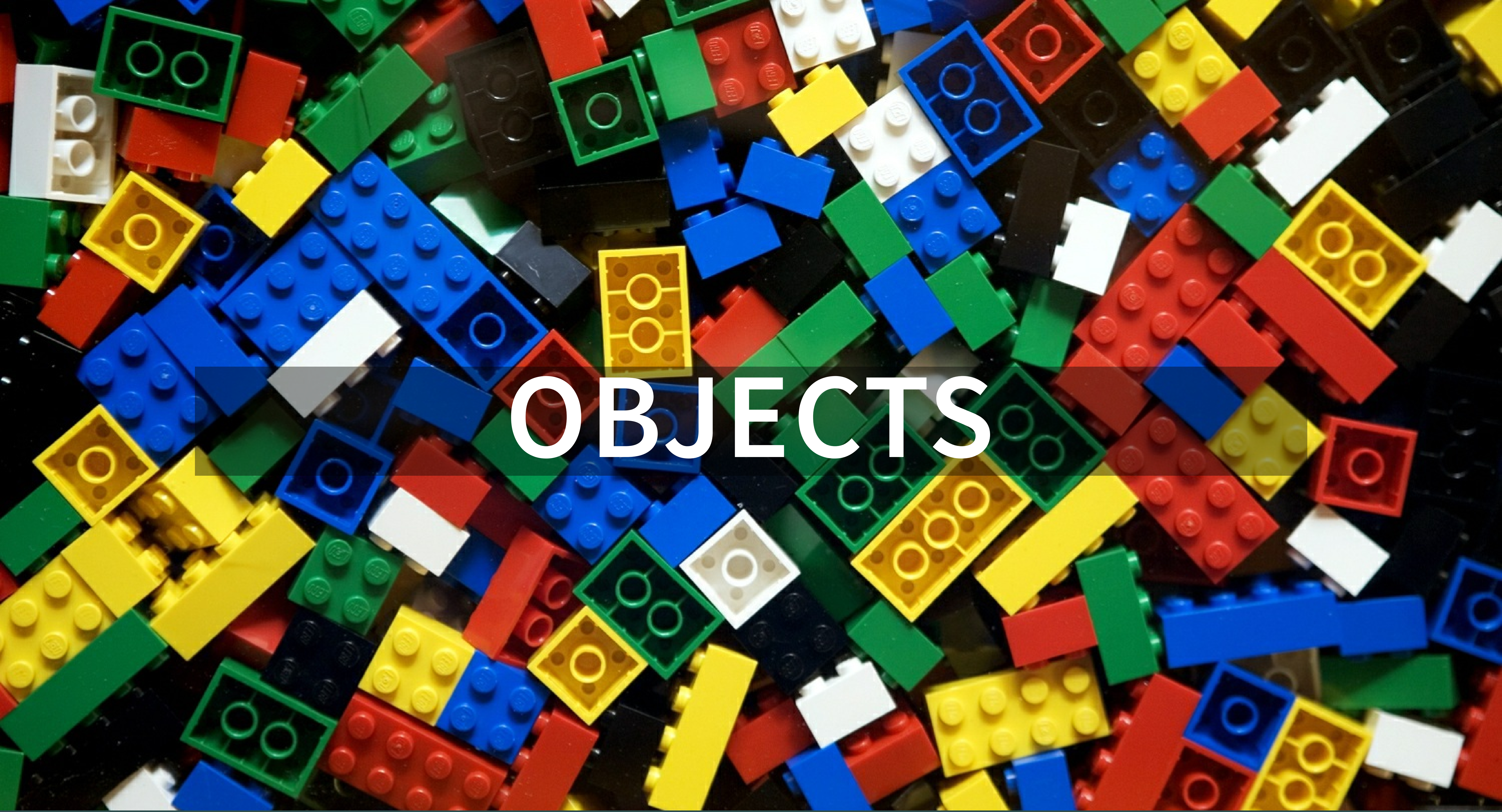
PICK SOME OBJECT AROUND YOU

now try to describe some of its defining attributes

and it's behaviour

A CLASS ATTEMPTS TO ACHIEVE A SIMILAR GOAL

but within the Python environment



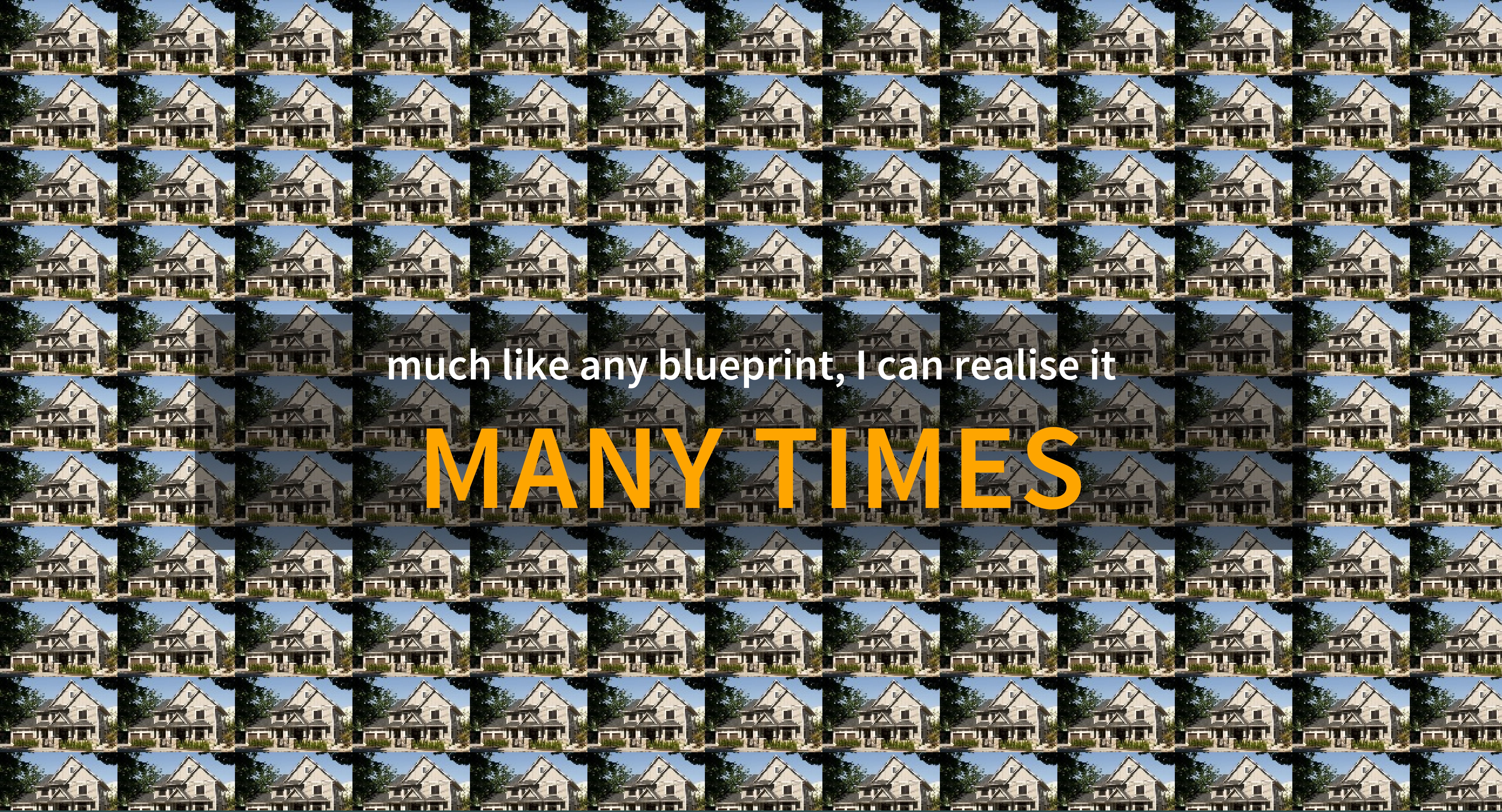
OBJECTS

OBJECTS

are **instances** of a class

i.e. they are a **realisation** of the *blueprint*

e.g. a house built from a set of defined plans



much like any blueprint, I can realise it

MANY TIMES

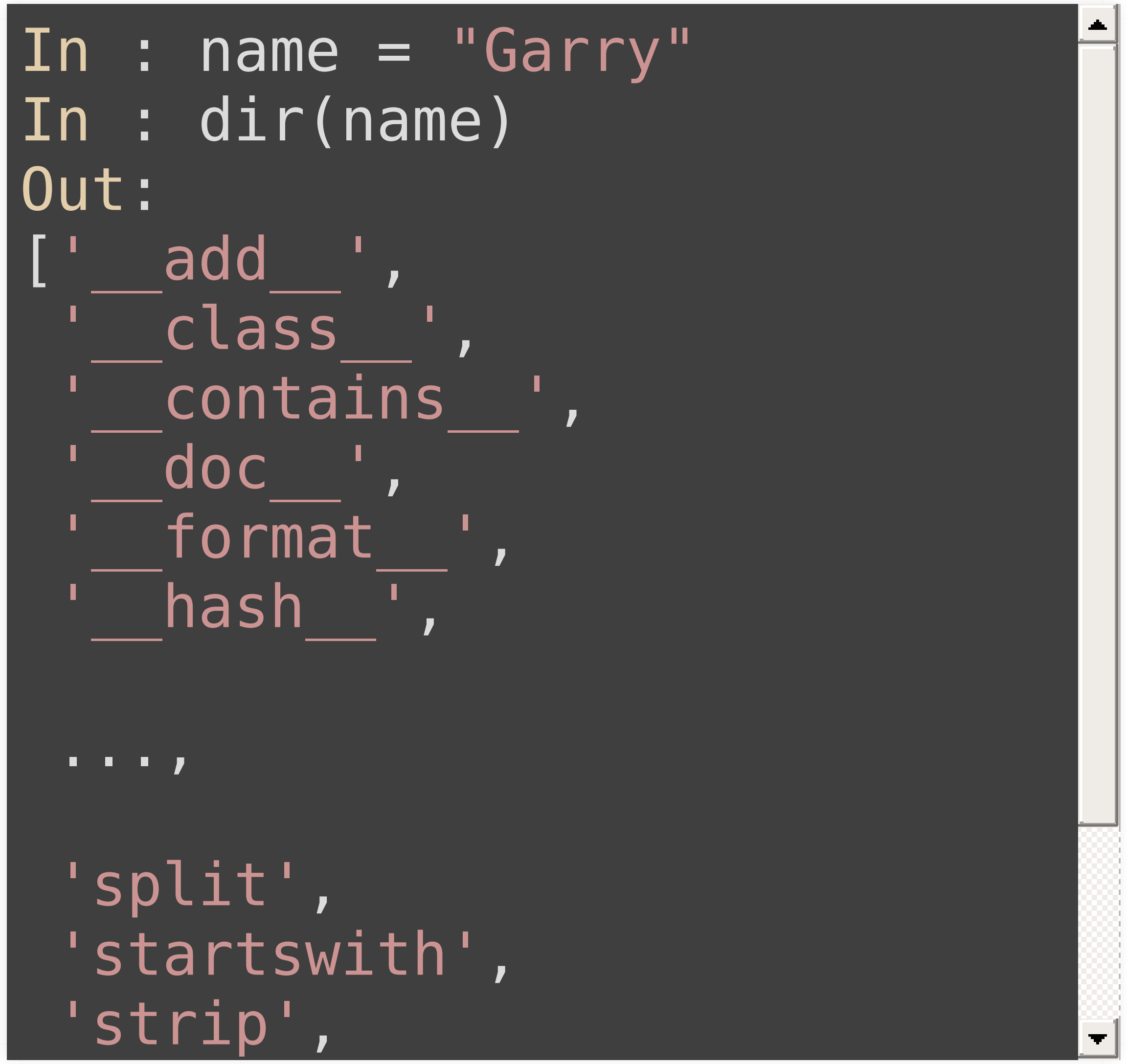
CLASSES ARE COMMONLY REFERRED TO AS

DATA STRUCTURES

and you've actually been using them since the first lesson

GOING BACK TO

the `string` data type



```
In : name = "Garry"
In : dir(name)
Out:
['__add__',
 '__class__',
 '__contains__',
 '__doc__',
 '__format__',
 '__hash__',

 ...,

 'split',
 'startswith',
 'strip',
```

`name` is a realisation of the "`string` class"

with methods and behaviours

and we can create as many strings as we want

SO LET'S CREATE SOME OF OUR OWN

DEFINING CLASSES IN PYTHON

Let's create one to describe a customer account with a bank

A CLASS-Y ACCOUNT

classes are defined like this

```python
class Account:

    # attributes defined here

    # behaviour defined here
```

and to create a realisation (object) of this class

```python
ac1 = Account()
```

the process of creating an object is called

INSTANTIATION

A CLASS-Y ACCOUNT

```python
class Account:
    """
    Data structure to describe a customer with the business
    """
    def __init__(self, name, deposit):
        # customer attributes
        self.name = name
        self.savings = deposit

    def account_detail(self):
        """
        Print current financial position of customer
        """
        message = "Customer: %s has $%i" % (self.name, self.savings)
        print(message)
```

now let's put it to action!

A CLASS-Y ACCOUNT

```
In : ac1 = Account("Peter", 1000)

In : ac1
Out: <__main__.Customer at 0x7f605c04e550>

In : ac1.name
Out: "Peter"

In : ac1.savings
Out: 1000

In : ac1.account_detail()
Out: "Customer: Peter has $1000"

In : ac1.account_detail
Out: <bound method Account.account_detail of
        <__main__.Account object at 0x7f605c04e550>>
```

A CLASS-Y ACCOUNT

```python
class Account:
    """
    Data structure to describe a customer with the business
    """
    def __init__(self, name, deposit):
        # customer attributes
        self.name = name
        self.savings = deposit

    def account_detail(self):
        """
        Print current financial position of customer
        """
        message = "Customer: %s has $%i" % (self.name, self.savings)
        print(message)
```

there are a couple of things that will be new to you

`self`

means it is referencing itself

↓

i.e. when the object is created, such attributes and methods will automatically refer to it

```
ac1 = Account("Peter", 1000)

# this is the same
ac1.account_detail()

# as this
Account.account_detail(ac1)
```

__INIT__

two things are going on here

1. when an object is created, variables (and their values) are automatically assigned to the object

2. the leading and trailing double underscore, refer to a **reserved** Python method

↓

i.e. Python knows what to do when it sees it

A CLASS-Y ACCOUNT

Having created this new data structure, I can create as many as I want!

```
ac1 = Account("Mathew", 2003)
ac2 = Account("Mark", 4000)
ac3 = Account("Luke", 1350)
ac4 = Account("John", 10420)
ac5 = Account("Judas", 30)
```

and do it in a more *pythonic* way

```
disciples = [("Mathew", 2003), ("Mark", 4000), ("Luke", 1350), \
                ("John", 10420), ("Judas", 30)]
accounts = []

# add customer information to accounts list
for (name, deposit) in disciples:

    acc_obj = Account(name, deposit)
    accounts.append(acc_obj)
```

A CLASS-IER ACCOUNT

```
class Account:
    """
    Data structure to describe a customer bank account
    """
    num_customers = 0

    def __init__(self, name, deposit):
        # attributes
        self.name = name
        self.savings = deposit
        # keep track of customers
        Account.num_customers += 1
        self.accnum = Account.num_customers

    def account_detail(self):
        """
        Print current financial position of customer
        """
        message = "Customer: %s [AC: %s] has $%i" \
                            % (self.name, self.accnum, self.savings)
        print(message)
```

A CLASS-IER ACCOUNT

```
In : ac1 = Account("John", 10420)

In : ac1.account_detail()
Out: "Customer: John [AC: 1] has $10420"

In : ac2 = Account("Judas", 30)

In : ac2.account_detail()
Out: "Customer: Judas [AC: 2] has $30"
```

Every time we create a new Account object

an account number is assigned according to the counter

AN EVEN CLASS-IER ACCOUNT

```python
class Account:
    """
    Data structure to describe a customer bank account
    """
    num_customers = 0

    def __init__(self, name, deposit):
        # attributes
        self.name = name
        self.savings = deposit
        # keep track of customers
        Account.num_customers += 1
        self.accnum = Account.num_customers

    def account_detail(self):
        """
        Print current financial position of customer
        """
        message = "Customer: %s [AC: %s] has $%i" \
                          % (self.name, self.accnum, self.savings)
        print(message)

    def compound_interest(self, rate, freq, term):
        """
        Calculates compound interest
        """
        new_pos = self.savings*(1 + rate/freq)**(freq*term)

        return new_pos
```

AN EVEN CLASS-IER ACCOUNT

```
In : ac1 = Account("Mark", 4000)

In : ac1.account_detail()
Out: "Customer: Mark has $4000"

In : ac1.compound_interest(0.05, 4, 10)
Out: 5154.7267
```

However, we are not changing the state of the object

```
In : ac1.account_detail()
Out: "Customer: Mark has $4000"
```

the Class `method` is just returning a value

to change the state of the object

```python
class Account:
    """
    Data structure to describe a customer bank account
    """
    num_customers = 0

    def __init__(self, name, deposit):
        # attributes
        self.name = name
        self.savings = deposit
        # keep track of customers
        Account.num_customers += 1
        self.accnum = Account.num_customers

    def account_detail(self):
        """
        Print current financial position of customer
        """
        message = "Customer: %s [AC: %s] has $%i" \
                    % (self.name, self.accnum, self.savings)
        print(message)

    def compound_interest(self, rate, freq, term):
        """
        Calculates compound interest
        """
        new_pos = self.savings*(1 + rate/freq)**(freq*term)

        return new_pos
```

to change the state of the object

```
class Account:
    """
    Data structure to describe a customer bank account
    """
    num_customers = 0

    def __init__(self, name, deposit):
        # attributes
        self.name = name
        self.savings = deposit
        # keep track of customers
        Account.num_customers += 1
        self.accnum = Account.num_customers

    def account_detail(self):
        """
        Print current financial position of customer
        """
        message = "Customer: %s [AC: %s] has $%i" \
                  % (self.name, self.accnum, self.savings)
        print(message)

    def compound_interest(self, rate, freq, term):
        """
        Calculates compound interest
        """
        self.savings = self.savings*(1 + rate/freq)**(freq*term)

        return self.savings
```

AN EVEN CLASS-IER ACCOUNT

```
In : ac1 = Account("Mark", 4000)

In : ac1.account_detail()
Out: "Customer: Mark has $4000"

In : ac1.compound_interest(0.05, 4, 10)
Out: 5154.7267

In : ac1.account_detail()
Out: "Customer: Mark has $5154.7267"
```

the method now directly interacts with the object

this kind of object manipulation leads to our next topic

METHOD CHAINING

or piping

WHAT IS CHAINING OR PIPING?

Originates from Unix

$$x \rightarrow f(x) \rightarrow g(x) \rightarrow h(x) \rightarrow y$$

where some data could be fed through a sequence of processes to change it

```
data | group | then sort | then average
```

CHAINING IN PYTHON

we can achieve the same behaviour with our class objects

```python
class Account:
    """
    Data structure to describe a customer bank account
    """
    # other stuff usually here

    def compound_interest(self, rate, freq, term):
        """
        Calculates compound interest
        """
        self.savings = self.savings*(1 + rate/freq)**(freq*term)

        return self.savings
```

we just need to make a slight change to our code

CHAINING IN PYTHON

we can achieve the same behaviour with our class objects

```
class Account:
    """
    Data structure to describe a customer bank account
    """
    # other stuff usually here

    def compound_interest(self, rate, freq, term):
        """
        Calculates compound interest
        """
        self.savings = self.savings*(1 + rate/freq)**(freq*term)

        return self
```

we just need to make a slight change to our code

LET'S ADD SOME OTHER METHODS

```python
class Account:
    """
    Data structure to describe a customer bank account
    """
    # other stuff usually here

    def withdraw(self, amount):
        """
        Charge an account keeping fee
        """
        self.savings -= amount

        return self

    def deposit(self, amount):
        """
        Increase savings by deposit amount
        """
        self.savings += amount

        return self
```

AND CHAIN THESE TOGETHER

```
In : ac1 = Account("Mark", 4000)

In : ac1.account_detail()
Out: "Customer: Mark has $4000"

In : ac1.compound_interest(0.05, 4, 10).withdraw(500).deposit(100)

In : ac1.account_detail()
Out: "Customer: Mark has $4854"

In : ac1.savings
Out: 4854
```

we use the `dot` notation to connect the methods together

NOW, JUST ONE MORE THING TO LOOK AT WITH CLASSES

BEHAVIOUR

notice with `string` data types

WE HAVE THIS BEHAVIOUR

```
In : string1 = "Hello "
In : string2 = "world!"

In : string1 + string2
Out: "Hello world!"
```

the addition symbol `+` is used to join two string together as one

under the hood, Python is joining the objects together

WE CAN ADD SIMILAR BEHAVIOUR TO OUR OWN OBJECTS

for instance, imagine I have two customers

Mark	Judas

who want to create a join account

↓

Can I give my **Account objects** behaviour to do this?

A FIRST CLASS ACCOUNT

```python
class Account:
    """
    Data structure to describe a customer bank account
    """
    num_customers = 0

    def __init__(self, name, deposit):
        # attributes
        self.name = name
        self.savings = deposit
        # keep track of customers
        Account.num_customers += 1
        self.accnum = Account.num_customers

    def __add__(self, other):

        self.joint_name = "%s and %s" % (self.__name, other.__name)
        self.joint_savings = self.__savings + other.__savings

        return Account(self.joint_name, self.joint_savings)

    # other stuff usually here
```

Let's go through this step-by-step

A FIRST **CLASS** ACCOUNT

First, create two different accounts

```
In : ac1 = Account("Mark", 4000)
In : ac1.account_detail()
Out: "Customer: Mark [AC: 1] has $4000"

In : ac2 = Account("Judas", 30)
In : ac2.account_detail()
Out: "Customer: Judas [AC: 2] has $30"
```

Now add them together to create a joint account

```
In : ac3 = ac1 + ac2
Out: "Customer: Mark and Judas [AC: 3] has $4030"
```

How cool is that!

FINALLY

let's have our account object be represented by something

```
class Account:
    """
    Data structure to describe a customer bank account
    """
    num_customers = 0

    def __init__(self, name, deposit):
        # attributes
        self.name = name
        self.savings = deposit
        # keep track of customers
        Account.num_customers += 1
        self.accnum = Account.num_customers

    def __repr__(self):
        state = "ACC: %s | NAME: %s | SAVINGS: %s" \
                    % (self.__accnum, self.__name, self.__savings)
        return state
```

A FIRST CLASS ACCOUNT

Our objects now have a direct output!

```
In : ac1 = Account("Mark", 4000)
In : print(ac1)
Out: "ACC: 1 | NAME: Mark | SAVINGS: 4000"

In : ac2 = Account("Judas", 30)
In : print(ac2)
Out: "ACC: 2 | NAME: Judas | SAVINGS: 30"

In : ac3 = ac1 + ac2
In : print(ac3)
Out: "ACC: 3 | NAME: Mark and Judas | SAVINGS: 4030"

In : ac2.compound_interest(0.05, 4, 10)
Out: "ACC: 2 | NAME: Judas | SAVINGS: 49.31"

In : ac1.withdraw(100).deposit(500)
Out: "ACC: 1 | NAME: Mark | SAVINGS: 4400"
```

How cool is that!

SUMMARY

CLASSES ARE INCREDIBLY POWERFUL

with objects:

✔ nearly anything can be conceived as a data structure

✔ we can group related data as an abstract type

✔ which can act and behave like any other Python variable

SOME CONVENTIONS TO REMEMBER

Class names are generally given an UPPERCASE title

♦

alternatively objects are given lowercase names

♦

always have *docstrings* throughout the class definition

♦

keep it simple

we ♥ Classes

we ♥ & objects

we ♥ Python

NEXT TIME

BONUS ROUND

INHERITANCE

Classes being derived from other Classes!

think of it in terms of there being sub-classes, e.g.

tree	circle
PLANT	SHAPE

Here, the Account class

```
class Customer:
    """
    Data structure to describe bank customer
    """
    def __init__(self, name, dob)
        self.name = name
        self.dob = dob

class Account(Customer):
    """
    Data structure to describe a customer account
    """
    num_customers = 0

    def __init__(self, name, dob, deposit):
        # give the account object access to Customer attributes
        Customer.__init__(name, dob)

        # attributes
        self.savings = deposit
        # keep track of customers
        Account.num_customers += 1
        self.accnum = Account.num_customers
```

inherits the attributes and behaviours of Customer

For one class to inherit another you need to pass it like so

```
class Account(Customer):
```

so `Account` will have the `name` and `dob` attributes of `Customer`

```
In : ac1 = Account("Judas", "03-04-30BC", 30)

In : ac1.dob
Out: "03-04-30BC"
```

even though they are defined separately

a good way to think about this is as a

PARENT-CHILD RELATIONSHIP

where the child inherits everything from the parent

consequently, we can break Classes down into a chain of relations

e.g.

`Bank → Customer → Account`

`Animal → Fish → Shark`

`Forest → Tree → Leaf`

and so forth