Objects and Classes 101

Object-Oriented Programming in Python

Why?

Why?

- Organize your code logically / like putting your code into drawers
- So other people or you 2 months from now having to work with your code have an easier time understanding the code
- Remain sane
- But OOP isn’t the only way

“Objects are like people. They’re living, breathing things that have knowledge inside them about how to do things and have memory inside them so they can remember things. And rather than interacting with them at a very low level, you interact with them at a very high level of abstraction, like we’re doing right here.”

–Steve Jobs

“Objects are like people. They’re living, breathing things that have knowledge inside them about how to do things and have memory inside them so they can remember things. And rather than interacting with them at a very low level, you interact with them at a very high level of abstraction, like we’re doing right here.”

–Steve Jobs


25 LBS
16
15
14
13
12

Some Terminology

- objects - a thing that has different methods you can call
- methods - an ability (function) that an object has
- classes - the definition/blueprint/template for a class of objects
- constructors (dunder init) - the method that constructs an
- instance variables (attributes) - a variable stored on an object
- inheritance - a class A inheriting another class B gets all the methods of B

You've Used Objects Before

```
cell_number = phonebook_dict.get('cell')
```

```
for name, phoneinfo in phonebook_dict.items():
    print name, phoneinfo
```

```
file_contents = myfile.read()
```

```
file_lines = myfile.readlines()
```

You've Used Objects Before

```
cell_number = phonebook_dict.get('cell')
```

```
for name, phoneinfo in phonebook_dict.items():
    print name, phoneinfo
```

```
file_contents = myfile.read()
```

Objects!!

```
file_lines = myfile.readlines()
```

Method Calls

```
cell_number = phonebook_dict.get('cell')
```

```
for name, phoneinfo in phonebook_dict.items():
    print name, phoneinfo
```

```
file_contents = myfile.read()
```

```
file_lines = myfile.readlines()
```

Objects!!

Method Calls!!

What is an object?

What is an object?

Functions + State

Objects as function + state

- Dictionaries have methods: `get`, `items`
- Dictionaries have state: the entries it needs to remember

Objects as function + state

- Lists have methods: `append, pop`
- Lists have state: the items in it that it has to remember

Objects as function + state

- Files have methods: `read, readline(), truncate()`
- Files have state: the current file pointer to the place in the file it has to read, the file open mode, whether the file is open or closed

Global Variable Counter Example

```
1   hello_count = 0
2
3   def say_hello():
4       global hello_count
5       print 'hello'
6       hello_count += 1
7
8   print 'Hello count %d' % hello_count
9   say_hello()
10  say_hello()
11  print 'Hello count %d' % hello_count
```

We have a global variable (state) and globals are nasty

Do that with an object

```
10    myhello = Hello()
11    print 'Hello count %d' % myhello.count
12    myhello.say_it()
13    myhello.say_it()
14    print 'Hello count %d' % myhello.count
```

The state is stored inside (or associated with) the object

We'll come back to this

```
10    myhello = Hello()
11    print 'Hello count %d' % myhello.count
12    myhello.say_it()
13    myhello.say_it()
14    print 'Hello count %d' % myhello.count
```

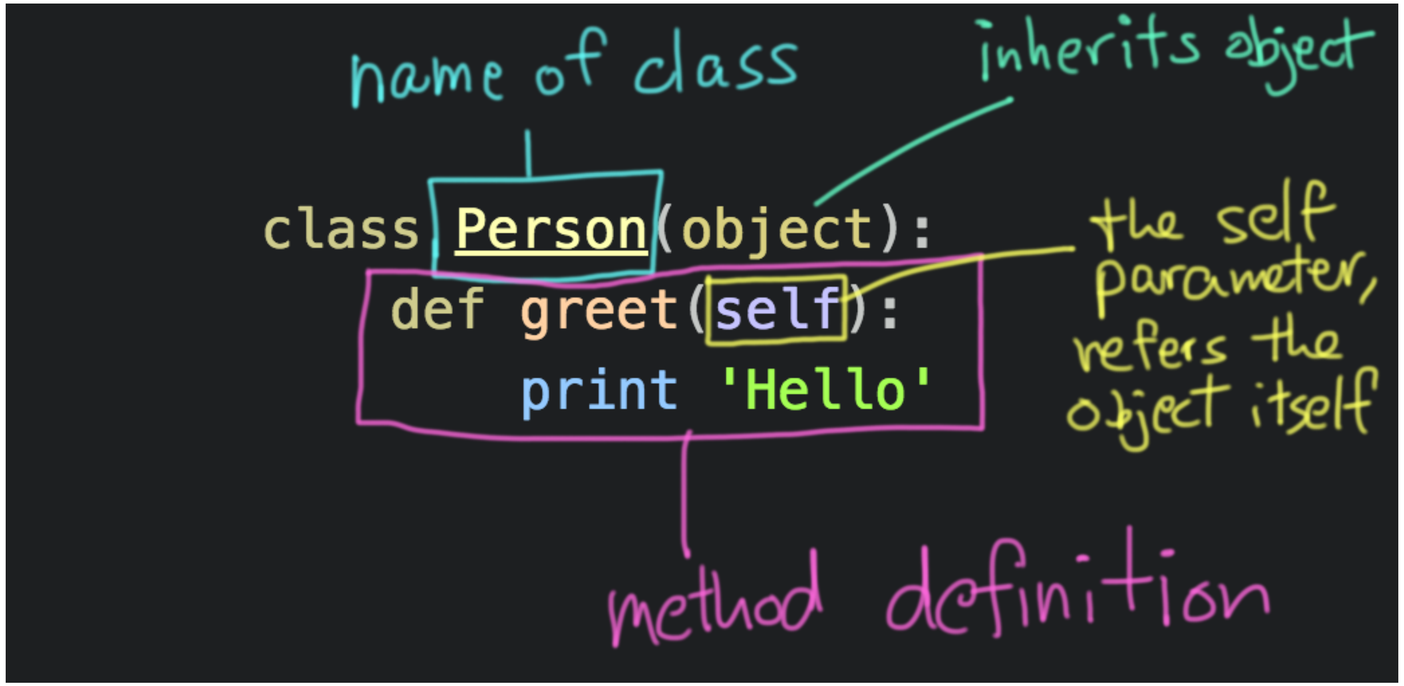
Let’s make a custom object

Let’s make a ***class***

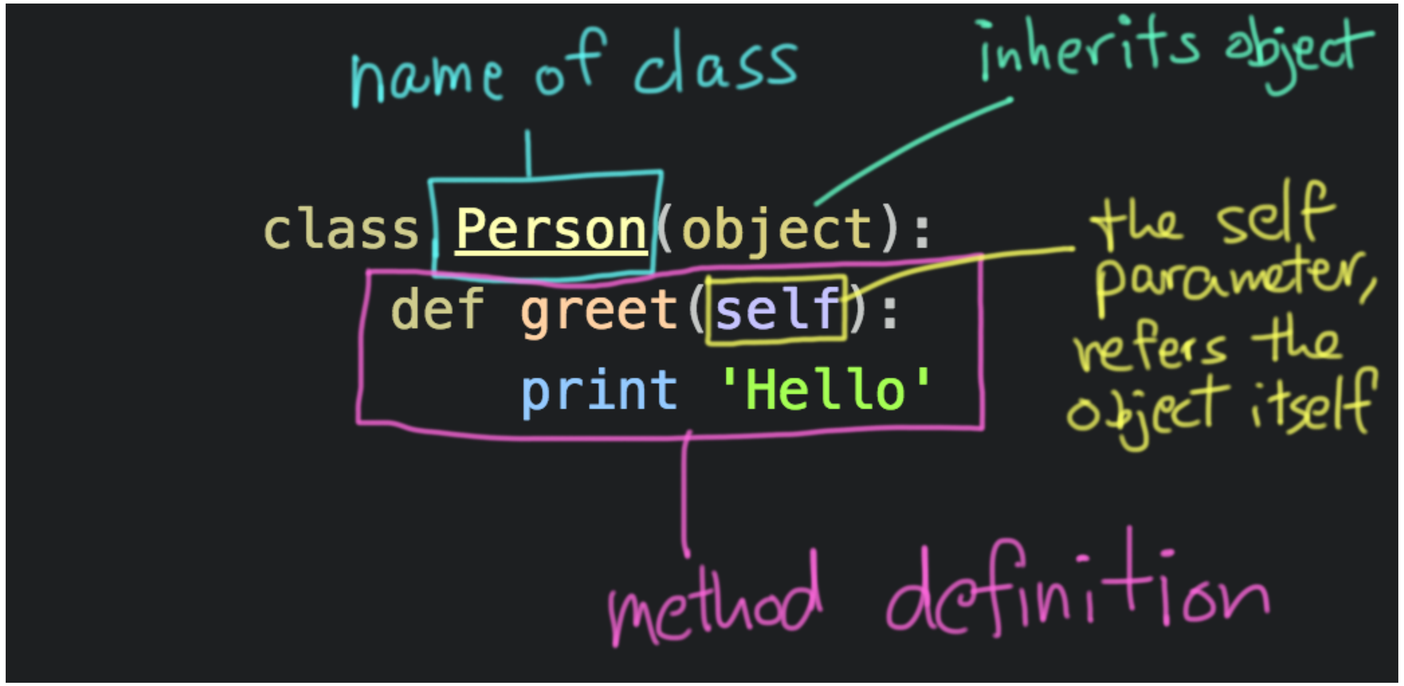
Make a class

```
class Person(object):
    def greet(self):
        print 'Hello'
```

Make a class


name of class
inherits object
class Person(object):
def greet(self):
print 'Hello'
the self parameter, refers the object itself
method definition

Make a class



The self parameter is weird, but you always need it

Create a Person object

Given the class `Person` (blueprint):

```
class Person(object):
    def greet(self):
        print 'Hello'
```

We can create an instance of a person object:

```
janice = Person()
```

Use a method of the object

Then we can call the **`greet`** method that's present on the person object:

```
janice = Person()
janice.greet()
```

That outputs "Hello"

```
class Person(object):
    def greet(self):
        print 'Hello'
```

Use a method of the object

Then we can call the **`greet`** method that's present on the person object:

```
janice = Person()
janice.greet()
```

That outputs "Hello"

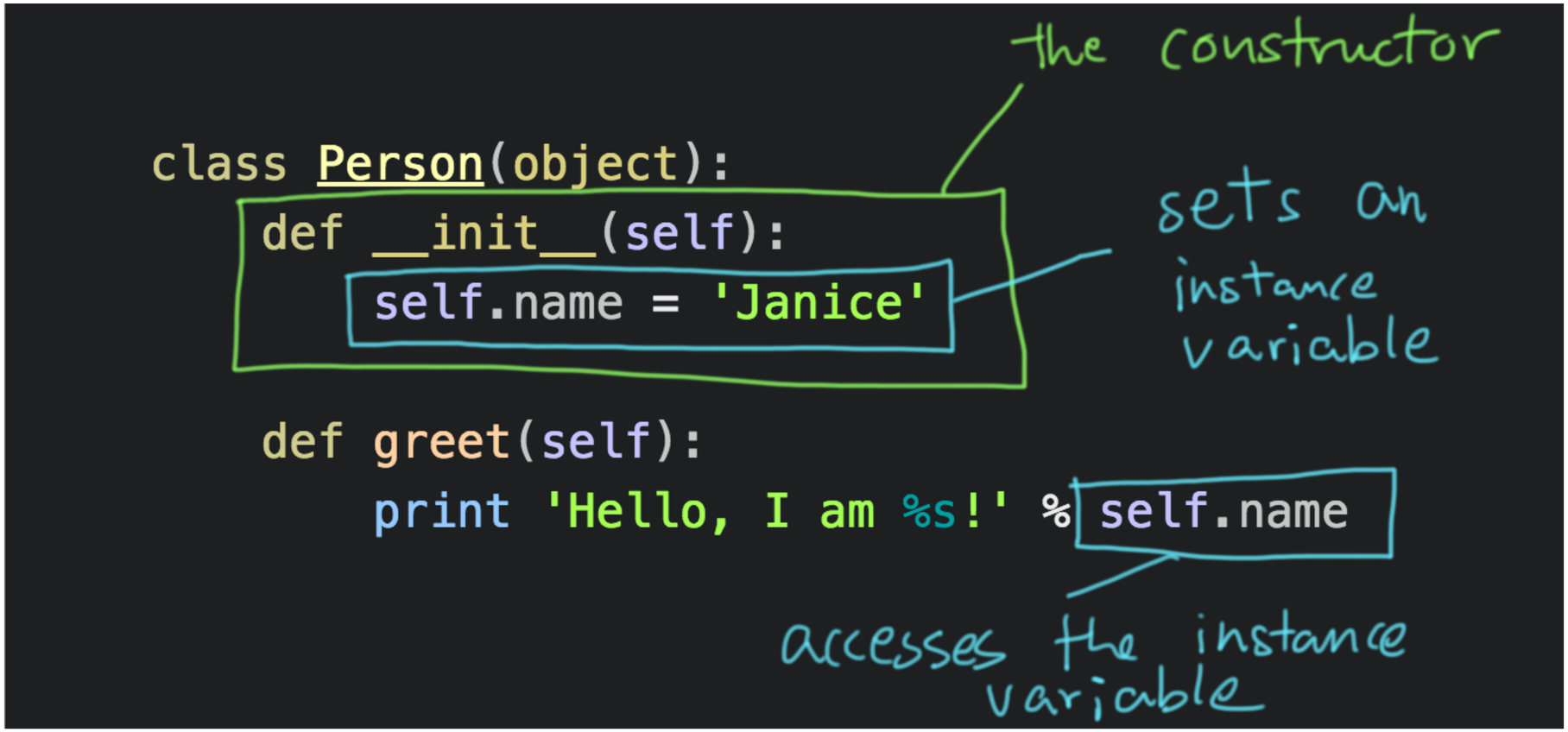
```
class Person(object):
    def greet(self):
        print 'Hello'
```

See that on Python Tutor

The Constructor and Instance Variables (a.k.a attributes)

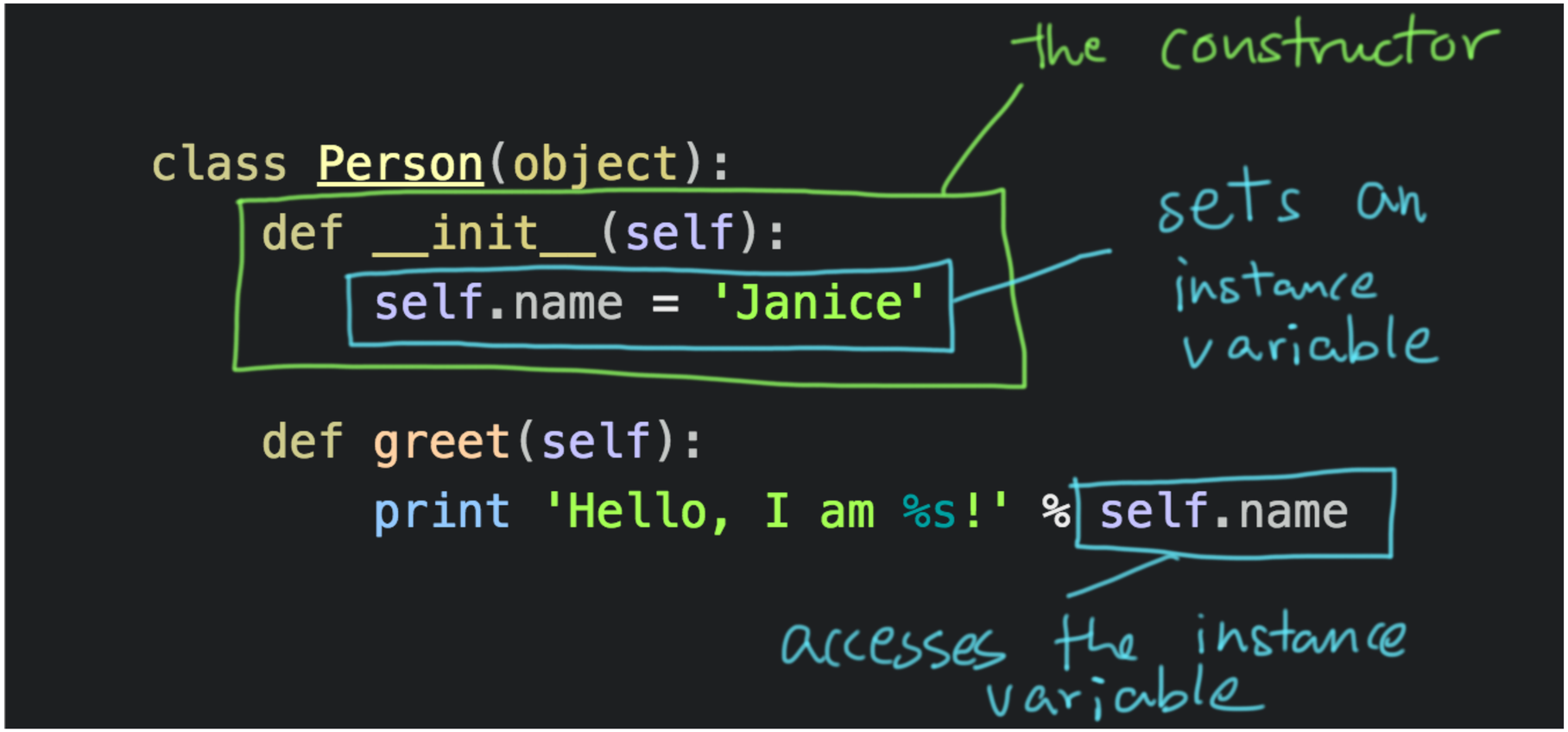
```
class Person(object):
    def __init__(self):
        self.name = 'Janice'

    def greet(self):
        print 'Hello, I am %s!' % self.name
```

The Constructor and Instance Variables (a.k.a attributes)



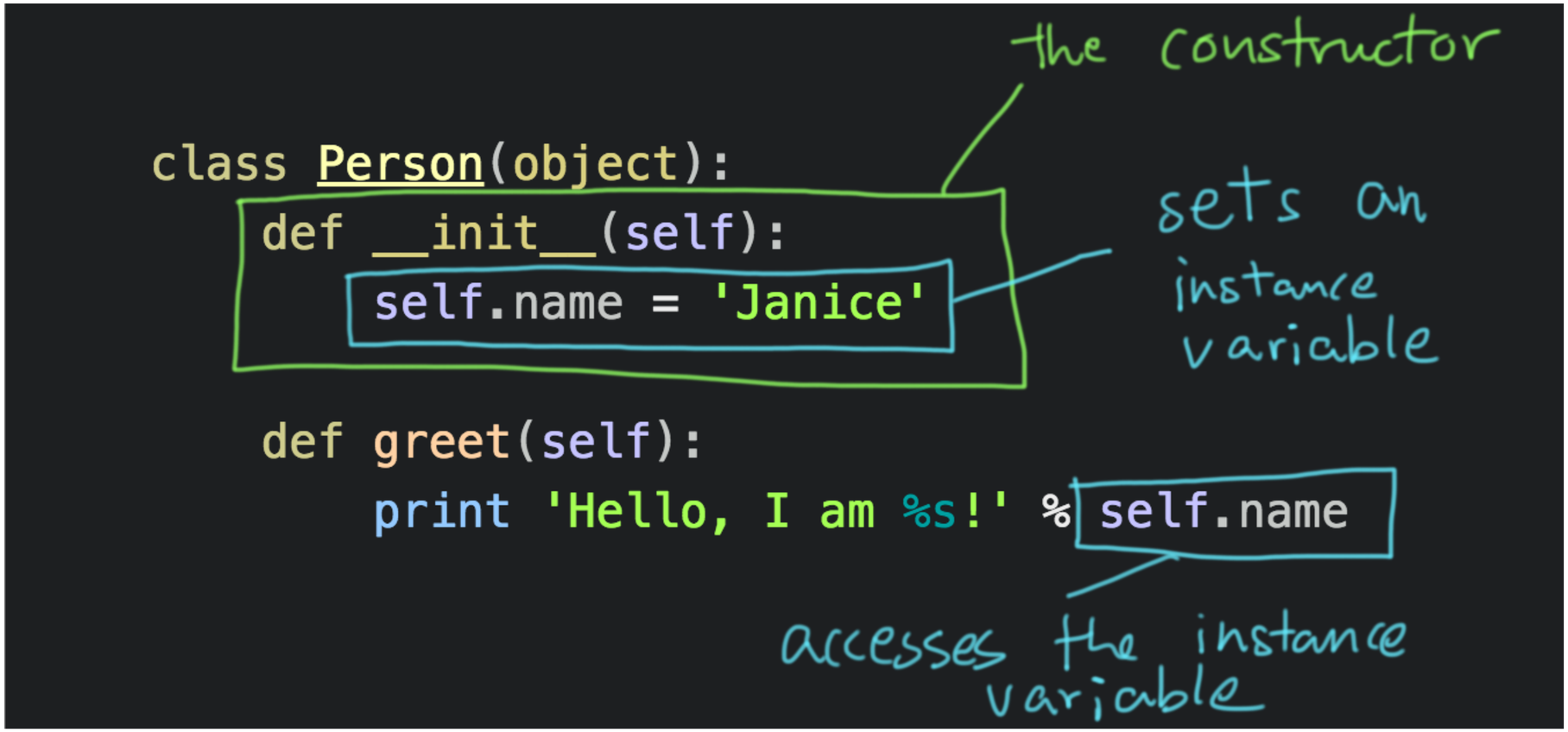
The constructor is called the moment a new instance of an object in created
it is used to set up initial state on the object

The Constructor and Instance Variables (a.k.a attributes)



`__init__` is pronounced as dunder init

The Constructor and Instance Variables (a.k.a attributes)



See that in Python Tutor

Now we come back to this

How to implement the **`Hello`** class?

```
10    myhello = Hello()
11    print 'Hello count %d' % myhello.count
12    myhello.say_it()
13    myhello.say_it()
14    print 'Hello count %d' % myhello.count
```

The `Hello` class

```
1    class Hello(object):
2        def __init__(self):
3            self.count = 0
4
5        def say_it(self):
6            print 'hello'
7            self.count += 1
```

`count` is initialized to `0` in the dunder init method
it is incremented in the `say_it` method.

Before and After

```
hello_count = 0

def say_hello():
    global hello_count
    print 'hello'
    hello_count += 1

print 'Hello count %d' % hello_count
say_hello()
say_hello()
print 'Hello count %d' % hello_count
```

```
class Hello(object):
    def __init__(self):
        self.count = 0

    def say_it(self):
        print 'hello'
        self.count += 1

myhello = Hello()
print 'Hello count %d' % myhello.count
myhello.say_it()
myhello.say_it()
print 'Hello count %d' % myhello.count
```

Back to this: Not all persons are Janice!

```
class Person(object):
    def __init__(self):
        self.name = 'Janice'

    def greet(self):
        print 'Hello, I am %s!' % self.name
```

Add a parameter

```
class Person(object):
    def __init__(self, name):
        self.name = name

    def greet(self):
        print 'Hello, I am %s!' % self.name
```

Add a parameter

```
class Person(object):
    def __init__(self, name):
        self.name = name

    def greet(self):
        print 'Hello, I am %s!' % self.name
```

Add a parameter

```
janice = Person('Janice')
janice.greet()
kareem = Person('Kareem')
kareem.greet()
```

Output

```
Hello, I am Janice!
Hello, I am Kareem!
```

Add a parameter to a method

```
class Person(object):
    def __init__(self, name):
        self.name = name

    def greet(self, other_person):
        print 'Hello %s, I am %s!' % (other_person.name, self.name)
```

We are going to add a parameter to the greet method as well.
That parameter is `other_person` and we will use it to greet that person by his/her name.

Add a parameter to a method

```
janice = Person('Janice')
kareem = Person('Kareem')
janice.greet(kareem)
kareem.greet(janice)
```

Output:

```
Hello Kareem, I am Janice!
Hello Janice, I am Kareem!
```

Now we can pass in a person to each call to greet,
so that Janice can greet Kareem and vice versa.

Note about method and constructor paremeters

```
class Person(object):
    def __init__(self, name):
        self.name = name

    def greet(self, other_person):
        print 'Hello %s, I am %s!' % (other_person.name, self.name)


    janice = Person('Janice')
    kareem = Person('Kareem')
    janice.greet(kareem)
    kareem.greet(janice)
```

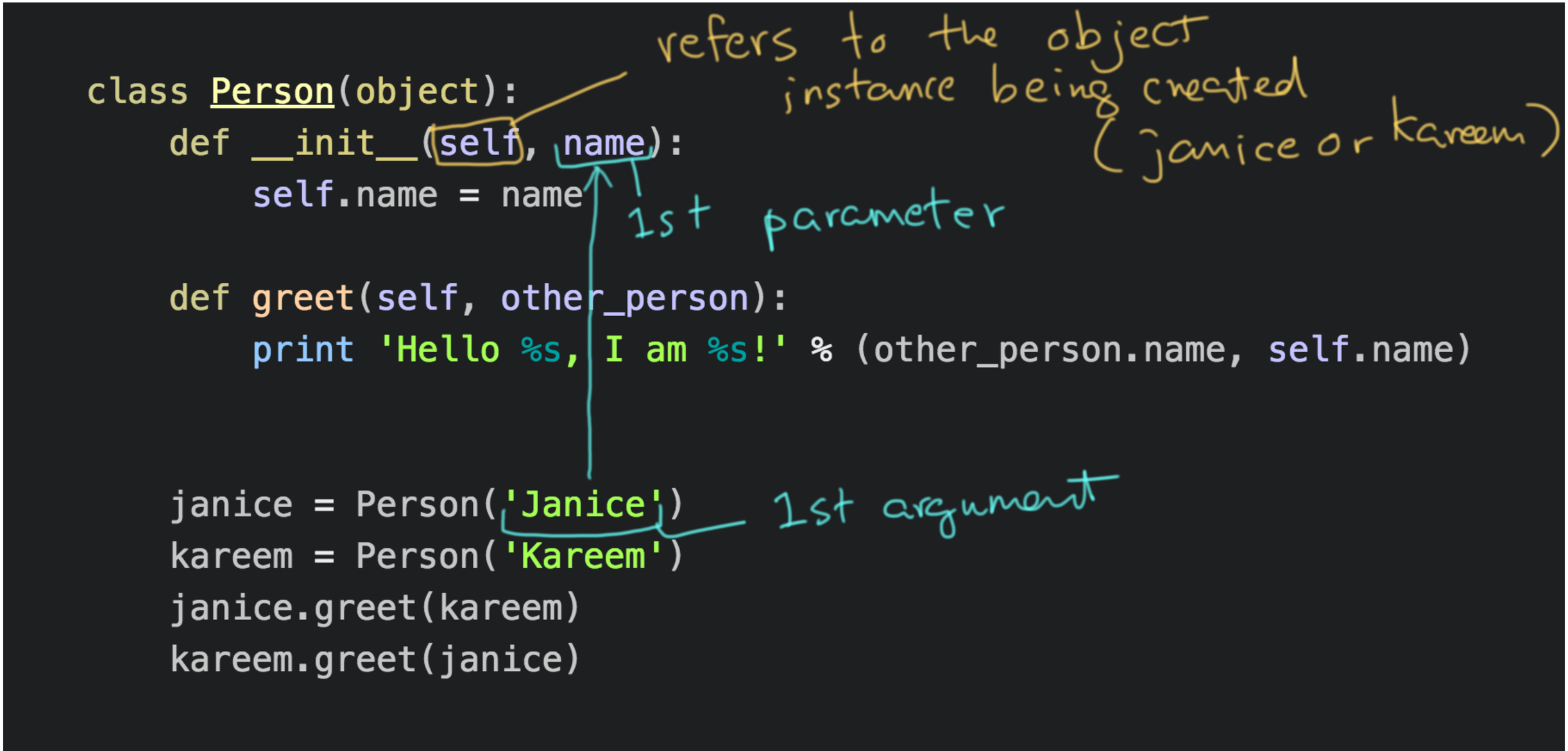
2 parameters

2 parameters

one argument

one argument

Note about method and constructor paremeters




```
class Person(object):
    def __init__(self, name):
        self.name = name

    def greet(self, other_person):
        print 'Hello %s, I am %s!' % (other_person.name, self.name)


    janice = Person('Janice')
    kareem = Person('Kareem')
    janice.greet(kareem)
    kareem.greet(janice)
```

Note about method and constructor paremeters

```
class Person(object):
    def __init__(self, name):
        self.name = name

    def greet(self, other_person):
        print 'Hello %s, I am %s!' % (other_person.name, self.name)


    janice = Person('Janice')
    kareem = Person('Kareem')
    janice.greet(kareem)
    kareem.greet(janice)
```

refers to the object instance whose greet method is being called

1st parameter

1st argument

Objects as Records

Dictionaries as Records

```
contact = {
    'first_name': 'Janice',
    'last_name': 'LaGrange',
    'email': 'janicel@yahoo.com',
    'phone': '485-2394-4934'
}

print 'First name: %s' % contact['first_name']
print 'Last name: %s' % contact['last_name']
print 'Email: %s' % contact['email']
print 'Phone: %s' % contact['phone']
```

Objects as Records

```
class Contact(object):
    def __init__(first_name, last_name, email, phone):
        self.first_name = first_name
        self.last_name = last_name
        self.email = email
        self.phone = phone

contact = Contact('Janice',
    'LaGrange', 'janicel@yahoo.com', '485-2394-4934')

print 'First name: %s' % contact.first_name
print 'Last name: %s' % contact.last_name
print 'Email: %s' % contact.email
print 'Phone: %s' % contact.phone
```