



# Python 3 Beginner

Chapter **00**

Previously, on Python 3 Beginner...

release 1.0.0

## What we saw last time

- ➡ We went on a tour of the Python Standard Library and learned how to:
  - ➡ Perform operations on the filesystem and paths
  - ➡ Read from and write to files
  - ➡ Handle base64-encoded data
  - ➡ Serialize/deserialize python objects to pickle and JSON
  - ➡ Use math functions and random number generators
  - ➡ Handle date and time data with ease in Python
  - ➡ And more...



# Q&A

# Where? Who?

## Class Material

► GitHub: <https://github.com/cstar-industries/python-3-beginner>

## Instructor

► Charles Francoise <[charles@cstar.io](mailto:charles@cstar.io)>



# Session Objectives

At the end of this session, you will be able to:

- ➡ Define your own types in Python with a **class**
- ➡ Add variables and methods to your custom types
- ➡ Create **instances** of your classes
- ➡ Use classes as a better replacement to other data types (like dicts)
- ➡ Understand and make use of **inheritance** to better re-use existing code
- ➡ Use **type annotations** and **docstrings** to make your code readable by anyone





# Session Syllabus

Chapter	Subject	Start	End	Total Time (in hours)
00	Previously on Python 3 Beginner...	14:00	14:15	00:15
01	Functions: Part I	14:15	15:00	00:45
	<i>Coffee Break</i>	15:00	15:10	00:10
02	Functions: Part II	15:10	15:55	00:45
	<i>Coffee Break</i>	15:55	16:05	00:10
03	Generators and Error Handling	16:05	16:50	00:45
	<i>Coffee Break</i>	16:50	17:00	00:10
	Workshop	17:00	18:00	01:00
	<b>Total</b>			<b>04:00</b>



# Python 3 Beginner

Chapter **01**

## Classes in Python

release 1.0.0

# Types are classes

- ➡ We've seen a number of types:
  - ➡ Some basic: int, float, bool...
  - ➡ Some more complex: list, dict, datetime...
- ➡ To get the type of an object we use the type built-in function

```
type(3)
```

```
int
```

```
print(type(3))
```

```
<class 'int'>
```

- ➡ What is this class ?

# Types are classes

► Classes have variables

```
print(datetime.resolution)
```

```
0:00:00.000001
```

► Classes have functions

```
print(datetime.now)
```

```
<function datetime.now>
```

```
print(datetime.now())
```

```
2020-04-25 21:55:25.754109
```

# Types are classes

► The objects' methods are functions from the class

```
d = datetime.now()  
print(datetime.timestamp(d))  
print(d.timestamp())
```

```
1587893892.289229  
1587893892.289229
```

```
s = 'hello'  
print(str.upper(s))  
print(s.upper())
```

```
HELLO  
HELLO
```

## Let's write some code!

```
s = 'hello'  
print(str.upper(s))  
print(s.upper())
```

```
HELLO  
HELLO
```

## Classes are namespaces

- ➡ To create a class, use the `class` keyword
- ➡ A class definition block (indented) begins after the colon
- ➡ As with functions, symbols defined inside the class are not visible outside the block

```
class A:  
    greeting = 'Hello'  
    counter = 0  
  
print(A)  
print(type(A))  
print(A.greeting)
```

```
<class '__main__.A'>  
<class 'type'>  
Hello
```



# Classes are namespaces

► Classes "contain" variables

```
print(A.counter)
A.counter += 1
print(A.counter)
```

```
0
1
```

► Classes "contain" functions

```
class A:
    def say_hello():
        print("Hello World!")

A.say_hello()
```

```
Hello World!
```

# Classes are namespaces

class A

greeting

'Hello World!'

say\_hello

print('Hello World!')

counter

0

# Let's write some code!

```
class A:  
    def say_hello():  
        print('Hello World!')
```

```
A.say_hello()
```

## Classes are types

- You can create an object using your class using a function-like notation

```
a = A()  
type(a)
```

```
__main__.A
```

- The class is the object's type

```
type(a) == A
```

```
True
```

- An object of type A is also called an *instance* of class A.



## Classes are types

➡ What happens when we use a function as a method?

```
a.say_hello()
```

```
TypeError: say_hello() takes 0 positional arguments but 1 was given
```

➡ Remember how we used the function from the class

```
s = 'hello'  
print(str.upper(s))  
print(s.upper())
```

```
HELLO  
HELLO
```

## Classes are types

- ➡ Let's define a new class B with a function taking a single argument: self

```
class B:  
    def say_hello(self):  
        print('Hello World!')  
  
b = B()  
b.say_hello()
```

Hello World!

- ➡ The function can still be used from the class

```
B.say_hello(b)
```

Hello World!

## Classes are types

- ➡ Using a method with the *instance.methodname* notation is equivalent to passing the instance as first argument to the function
- ➡ By convention, this first argument is always named `self`
- ➡ Similar to `this` in C++ and Java, but explicitly bound as an argument

# Classes are types

- ➡ Class attributes are available in instances

```
class C:  
    greeting = 'Hello!'  
  
print(C.greeting)  
  
c = C()  
print(c.greeting)
```

```
Hello!  
Hello!
```

## Let's write some code!

```
class B:  
    def print_self(obj):  
        print(obj)  
  
b = B()  
print(b)  
b.print_self()
```

```
<__main__.B object at 0x7ff4d0f840f0>  
<__main__.B object at 0x7ff4d0f840f0>
```

# Chapter Summary

In this chapter, we learned how to:

- ➡ Define our own **classes**
- ➡ Use them as types and create new **instances**
- ➡ Define variables and functions inside a class
- ➡ Call functions as **methods** of instances
- ➡ Understand the use of **self** in methods



# Python 3 Beginner

Chapter **02**

## Object-Oriented Programming

release 1.0.0

## More on methods

- Methods can have more than one argument

```
class A:  
    def add_two(self, num):  
        return num + 2  
  
a = A()  
print(a.add_two(8))
```

10

- First argument – self – always set to instance

## Initializer special function

- ➡ The special method `__init__` is called when you create a new instance
- ➡ Arguments given to the creation call are passed on to `__init__`
- ➡ Use the `__init__` function to create and initialize instance attributes

```
class A:  
    def __init__(self, id):  
        self.created_at = datetime.now()  
        self.id = id  
  
a = A(303)  
print(a.created_at)  
print(a.id)
```

2020-04-26 15:21:01.374870  
303

## Instance variables

- ▶ Instance variables are unique to each instance

```
a = A(303)

# Time passes...
b = A(123)

print(f'a({a.id}) - {a.created_at}')
print(f'b({b.id}) - {b.created_at}')
```

```
a(303) - 2020-04-26 15:21:01.374870
b(123) - 2020-04-26 16:20:11.459183
```

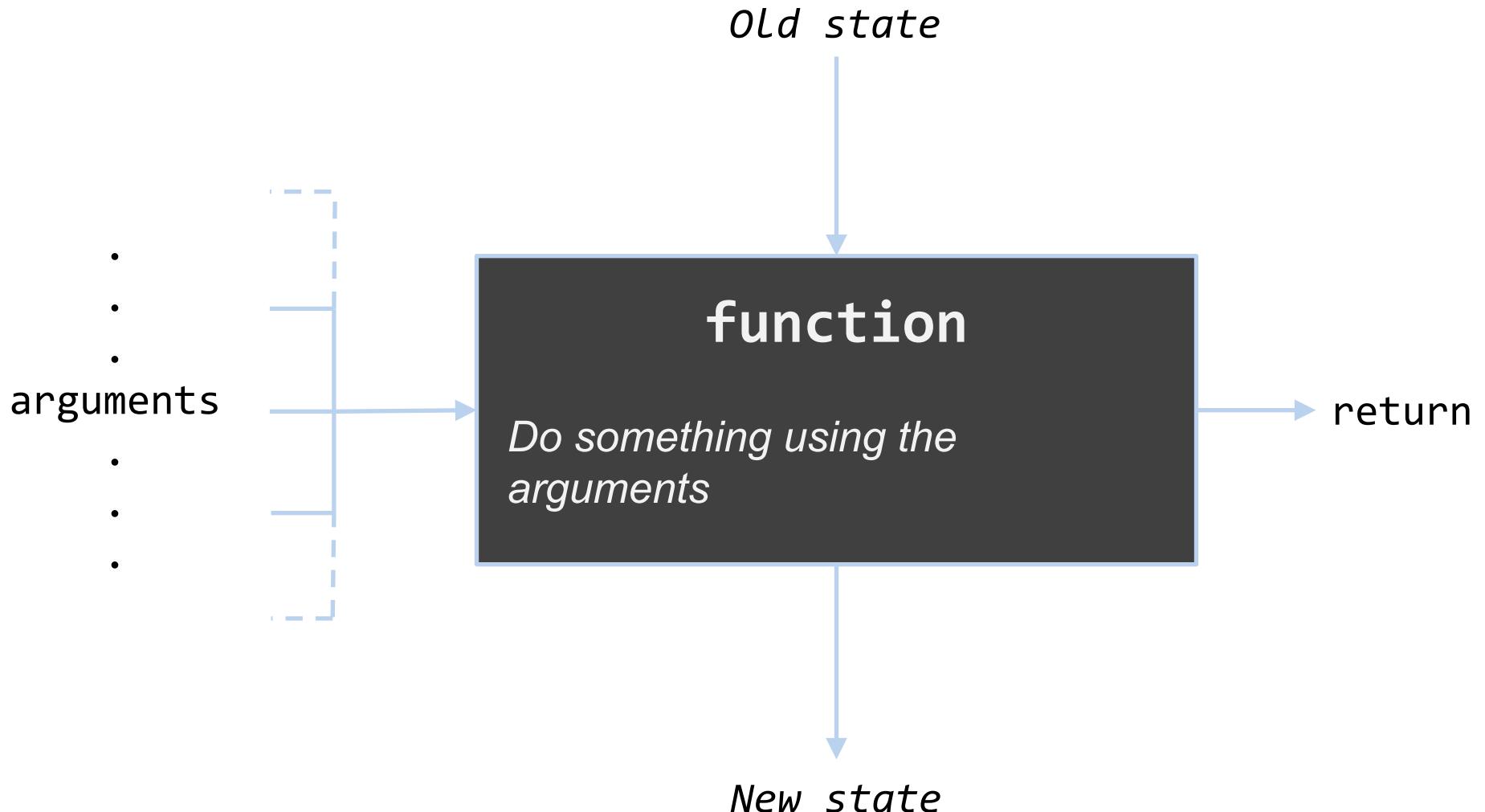
## Instance variables

→ Use methods to change instance attributes

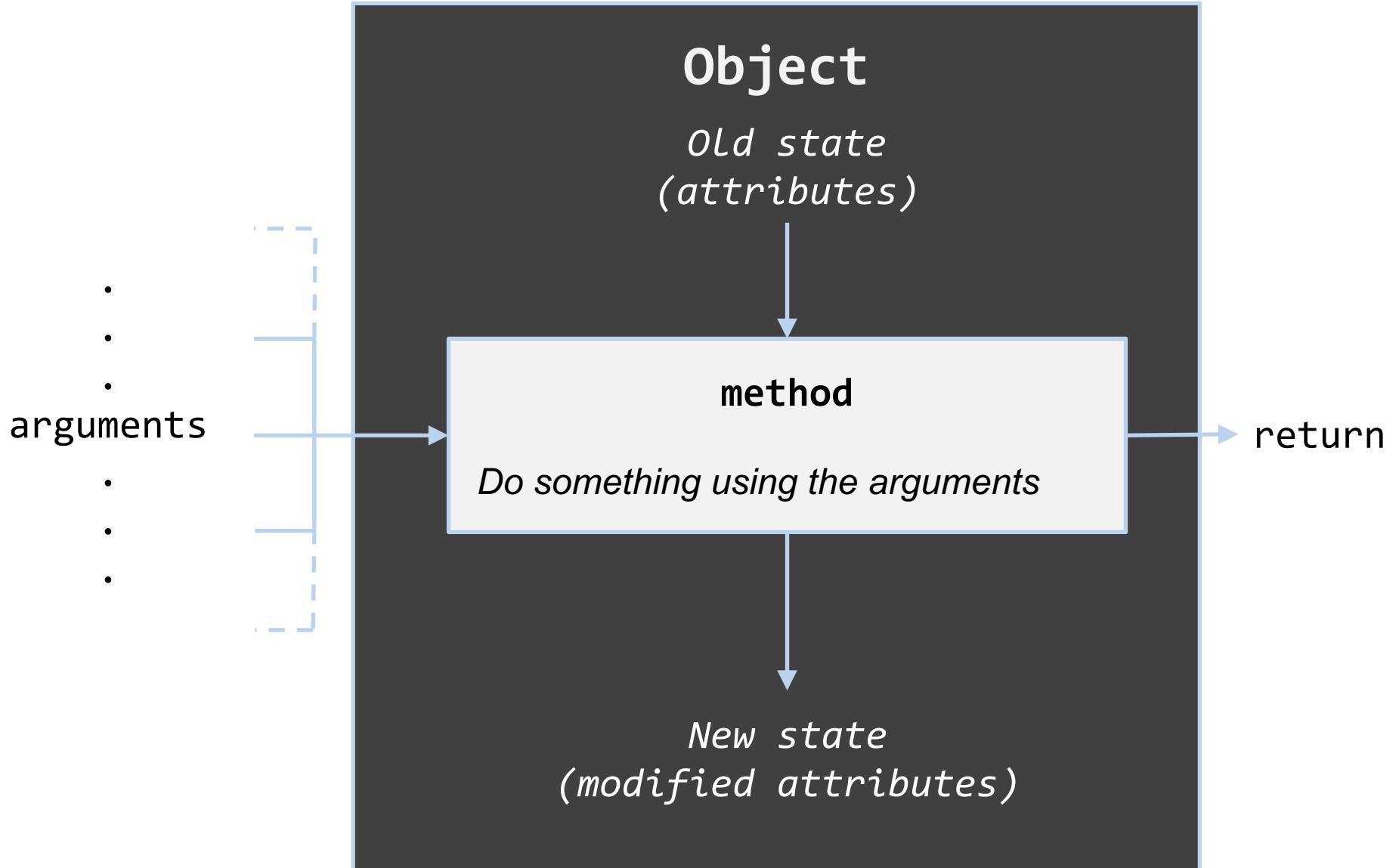
```
class Adder:  
    def __init__(self):  
        self.add_counter = 0  
  
    def add(self, a, b):  
        self.add_counter += 1  
        return a + b  
  
a1 = Adder()  
a2 = Adder()  
  
print(a1.add(2, 3), a1.add(3, 7))  
print(a1.add_counter)  
print(a2.add_counter)
```

```
5 10  
2  
0
```

## (Return to) Conceptual view of a function



# Conceptual view of an object



## Conceptual view of an object

*An object is a mechanism to bundle **state** (attributes) with **behavior** (methods).*

# Objects as higher level structured data

► Before classes we used dicts

```
guido = {'first_name': 'Guido', 'last_name': 'van Rossum', 'email':  
'guido@python.org'}  
  
def format_contact(c):  
    return f'{c["first_name"]} {c["last_name"]} <{c["email"]}>'  
  
print(format_contact(guido))
```

Guido van Rossum <guido@python.org>

► Let's do this with classes

# Objects as higher level structured data

- ➡ Write initializer
- ➡ Use attributes instead of dict keys
- ➡ Write functions as methods

```
class Contact:  
    def __init__(self, first_name, last_name, email):  
        self.first_name = first_name  
        self.last_name = last_name  
        self.email = email  
  
    def format(self):  
        return f'{self.first_name} {self.last_name} <{self.email}>'  
  
guido = Contact('Guido', 'van Rossum', 'guido@python.org')  
print(guido.format())
```

Guido van Rossum <guido@python.org>

# Let's write some code!

```
def f():
    a = []
    print(a)
    a += [0, 1, 2]
    print(a)

f()
print(a)
```

```
[]  
[0, 1, 2]
```

```
NameError: name 'a' is not defined
```

# Class attributes vs. Instance attributes

➡ Class attributes are shared by all instances

```
class A:  
    items = []  
  
    def add_item(self, item):  
        self.items.append(item)  
  
a = A()  
b = A()  
  
a.add_item('hello')  
b.add_item('world')  
print(a.items)
```

```
['hello', 'world']
```

# Class attributes vs. Instance attributes

➡ Instance attributes are unique to each instance

```
class A:  
    def __init__(self):  
        self.items = []  
  
    def add_item(self, item):  
        self.items.append(item)  
  
a = A()  
b = A()  
  
a.add_item('hello')  
b.add_item('world')  
  
print(a.items)
```

```
['hello']
```

## Class attributes vs. Instance attributes

➡ Class methods apply to the class object (not the instance)

```
class A:  
    number_of_instances = 0  
  
    def __init__(self):  
        self.add_instance()  
  
    @classmethod  
    def add_instance(cls):  
        cls.number_of_instances += 1  
  
a = A()  
b = A()  
  
print(A.number_of_instances)
```

2

## Class attributes vs. Instance attributes

➡ Static methods apply to no object (regular function)

```
class A:  
    def __init__(self):  
        self.add_instance()  
  
    @staticmethod  
    def greet(greetee):  
        print(f'Hello {greetee}!')  
  
A.greet('World')
```

Hello World!

## Let's write some code!

```
red_team = Team('Red Team')

red_team.add_person('Jack')
red_team.add_person('John')
red_team.add_person('Bart')

print(f'{red_team.name}: {red_team.people}')
```

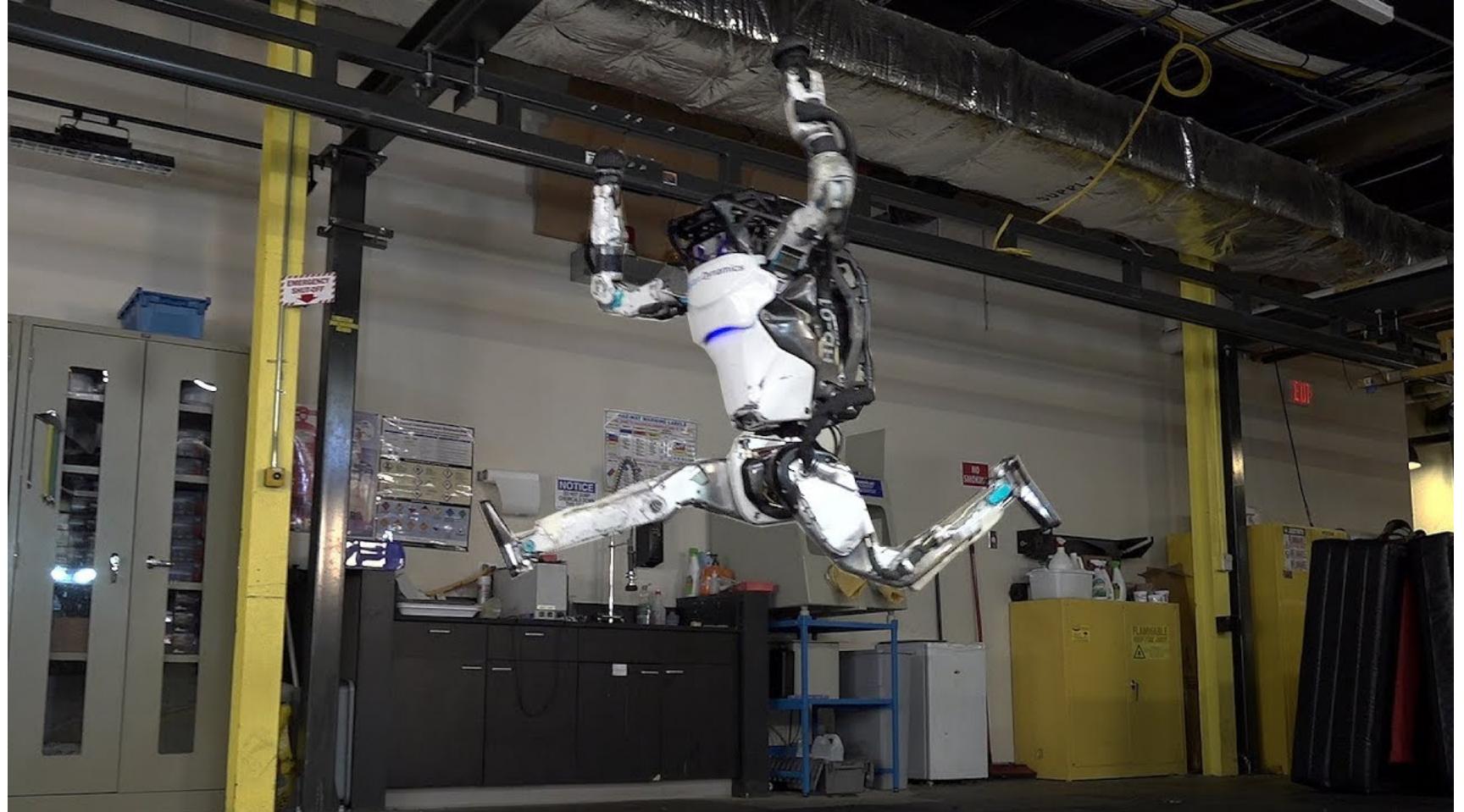
Red Team: ['Jack', 'John', 'Bart']

# Chapter Summary

In this chapter, we learned about:

- ➡ Initializing a class with the special `__init__` method
- ➡ Instance variables and Class variables
- ➡ The fundamental concept behind OOP
- ➡ Solving problems with classes

# Workout Time!





# Python 3 Beginner

Chapter **03**

Inheritance and Python specifics

release 1.0.0

## Inheritance

→ Sometimes, classes can share a lot of behavior with others

```
class Album:  
    def __init__(self, title, artist, songs):  
        self.title = title  
        self.artist = artist  
        self.songs = songs  
  
    def play(self):  
        print(f'Playing {self.songs[0]}...')  
  
class Playlist:  
    def __init__(self, title):  
        self.title = title  
        self.songs = []  
  
    def add_song(self, song):  
        self.songs.append(song)  
  
    def play(self):  
        print(f'Playing {self.songs[0]}...')
```

# Inheritance

➡ Create a class with all common code: the *superclass*

```
class Collection:  
    def __init__(self, title):  
        self.title = title  
        self.songs = []  
  
    def play(self):  
        print(f'Playing "{self.songs[0]}..."')
```

# Inheritance

➡ Create specialized classes (*subclasses*) without repeating code

```
class Album(Collection):
    def __init__(self, title, artist, songs):
        super().__init__(title)
        self.artist = artist
        self.songs = songs
```

```
class Playlist(Collection):
    def add_song(self, song):
        self.songs.append(song)
```

➡ Call methods from the superclass on the subclass

```
album = Album('2001', 'Dr. Dre', [...])
album.play()
```

Playing "Lolo(Intro)"...

## Let's write some code!

```
c = C('Instance of C', hello='world', count=1)
print(c.data)
```

```
Initializing C: Instance of C
Initializing B: Instance of C
Initializing A: Instance of C
{'hello': 'world', 'count': 1}
```

# Annotations

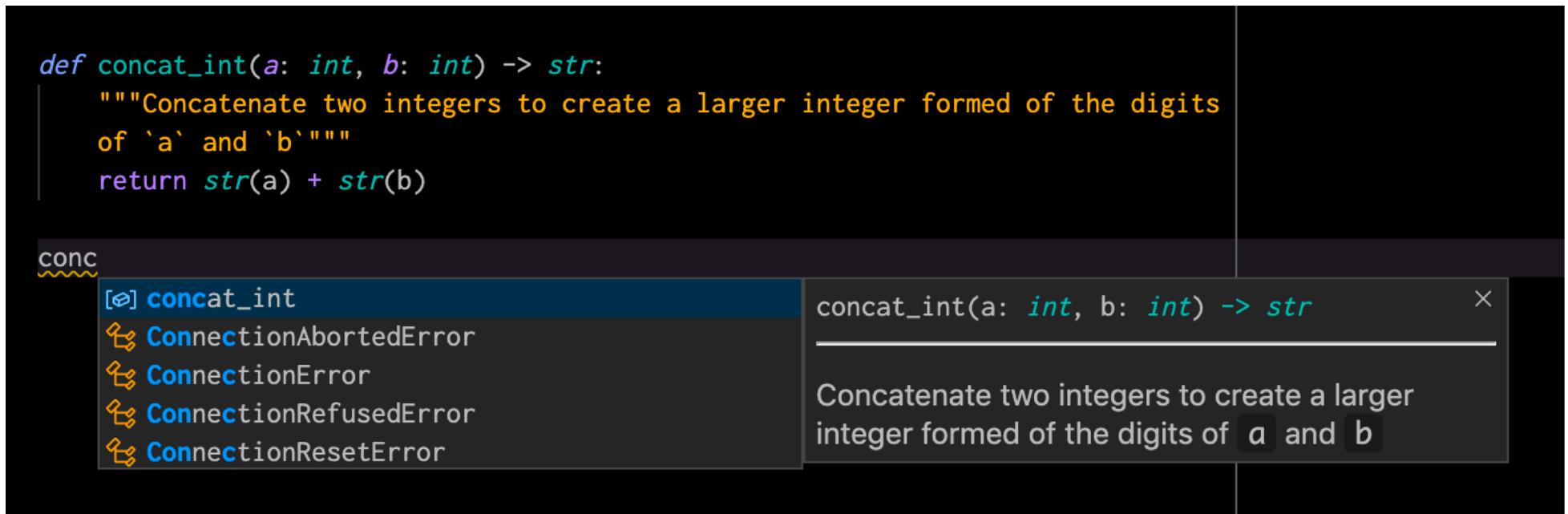
- Use *docstrings* to document your code (classes, attributes and methods)

```
class Accumulator:  
    """Accumulates count by adding values. Initialized at 0"""  
    def __init__(self):  
        self.acc = 0  
  
    def add(self, increment):  
        """Increase the count by increment"""  
        self.acc += increment  
  
help(Accumulator)
```

```
class Accumulator(builtins.object)  
|   Accumulates count by adding values. Initialized at 0  
|  
|   Methods defined here:  
|  
|       add(self, increment)  
|           Increase the count by increment
```

# Annotations

- ➡ Type annotations will be used by code editors and linters to help avoid bugs in your code



The screenshot shows a code editor with the following Python code:

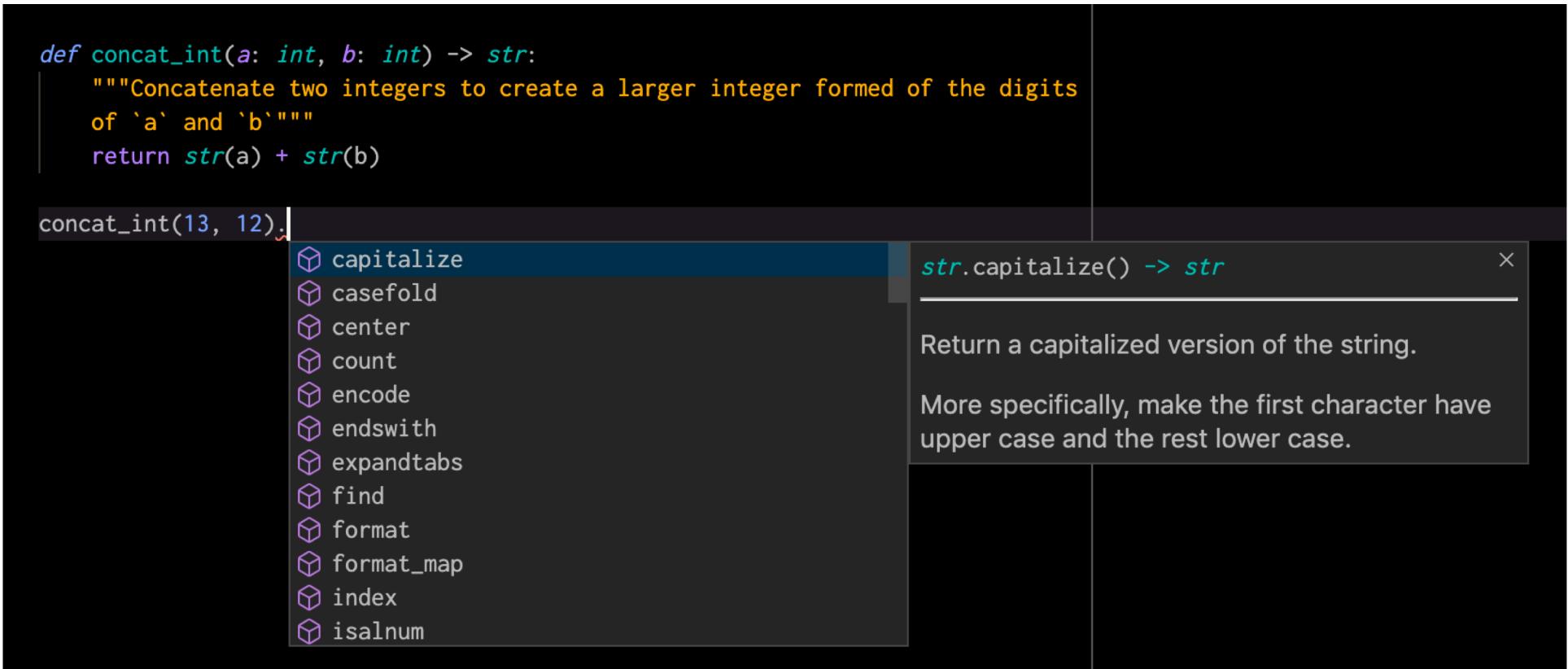
```
def concat_int(a: int, b: int) -> str:  
    """Concatenate two integers to create a larger integer formed of the digits  
    of `a` and `b`"""  
    return str(a) + str(b)
```

A tooltip is displayed over the word `concat`, showing the function definition and its docstring:

concat(a: int, b: int) -> str  
Concatenate two integers to create a larger integer formed of the digits of `a` and `b`

# Annotations

- ➡ Type annotations will be used by code editors and linters to help avoid bugs in your code



```
def concat_int(a: int, b: int) -> str:  
    """Concatenate two integers to create a larger integer formed of the digits  
    of `a` and `b`"""  
    return str(a) + str(b)  
  
concat_int(13, 12).|
```

The screenshot shows a code editor with a dark theme. A tooltip is open over the call to `str.capitalize()`. The tooltip contains the following text:

`str.capitalize() -> str`

Return a capitalized version of the string.  
More specifically, make the first character have upper case and the rest lower case.

## Special methods

- ➡ There are other special methods besides `__init__` to interact more idiomatically with the Python language

```
class Accumulator:  
    def __init__(self):  
        self.acc = 0  
  
    def add(self, increment):  
        self.acc += increment  
  
a = Accumulator()  
b = Accumulator()  
print(a == b)
```

False

## Special methods

► Redefine equality with `__eq__`

```
class Accumulator:  
    ...  
    def __eq__(self, other):  
        return self.acc == other.acc  
  
a = Accumulator()  
b = Accumulator()  
print(a == b)  
  
a.increment(1)  
print(a == b)
```

True  
False

## Special methods

► Provide relevant stringification with `__str__` and `__repr__`

```
class Accumulator:  
    ...  
    def __repr__(self):  
        return f'Accumulator({self.acc})'  
    def __str__(self):  
        return str(self.acc)  
  
a = Accumulator()  
a.add(1138)  
print(a)  
a
```

1138  
Accumulator(1138)

## Special methods

- ➡ Any other operator can be redefined:
  - ➡ < : `__lt__`, > : `__gt__`, != : `__ne__`...
- ➡ Indexing with [ ] can be defined in your classes: `__getitem__`, `__setitem__`...
- ➡ Type conversion with: `__str__`, `__int__`, `__bytes__`...
- ➡ Even iteration with for can work in your classes with: `__iter__` and `__next__`
- ➡ ! This power should not be abused. Only use this when you are sure it is useful, and the usage is unambiguous to anyone else.  
Remember: *Clear is better than clever.*
- ➡ To learn more, read the [documentation](#).

# Let's write some code!

```
class Accumulator:  
    """Accumulates count by adding values. Initialized at 0"""  
    def __init__(self):  
        self.acc = 0  
  
    def add(self, increment: int):  
        """Increase the count by increment"""  
        self.acc += increment  
  
    def __eq__(self, other):  
        return self.acc == other.acc  
    def __repr__(self):  
        return f'Accumulator({self.acc})'  
    def __str__(self):  
        return str(self.acc)
```

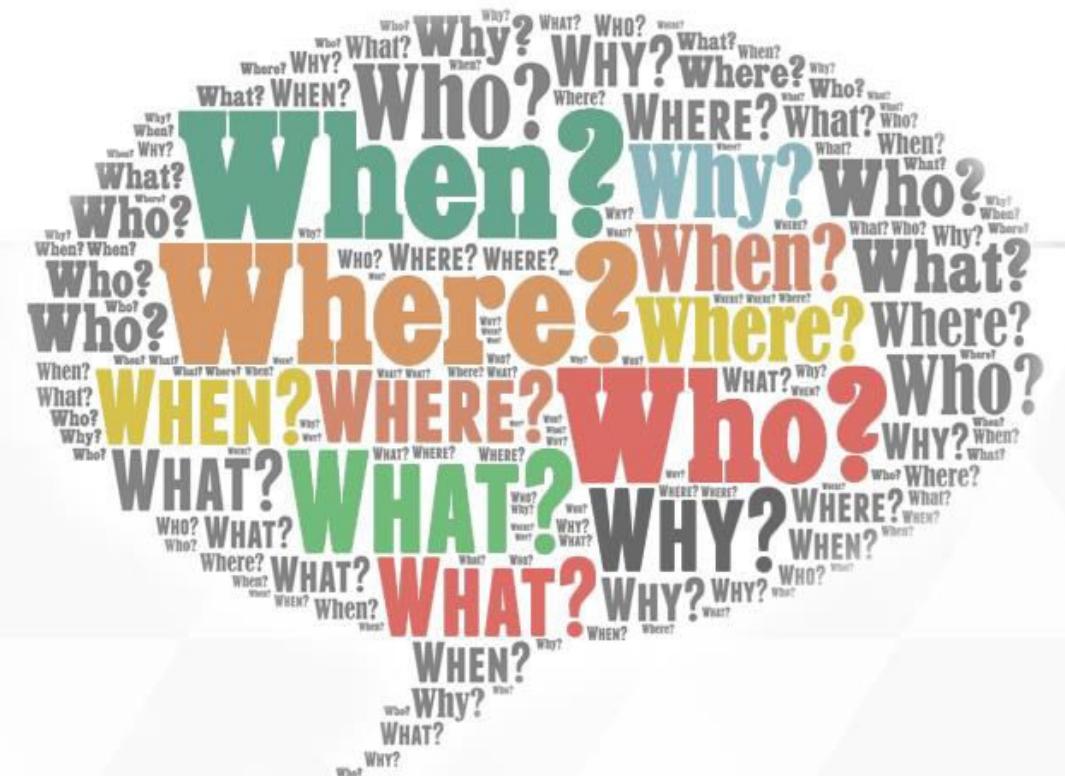
# Chapter Summary

In this chapter, we learned:

- ➡ The object-oriented concept of **inheritance** and how to implement it in Python
- ➡ How to document code effectively with **docstrings**
- ➡ How to use **type annotations** to document functions and facilitate code readability
- ➡ The special methods used to make our classes play better with Python

# Workout Time!





# Q&A

# Workshop





**Thank you for  
your attention!**